MASTER EN SCIENCES INFORMATIQUES

Contribution à l'adoption des bases de données NoSQL

Lamock, Martial

Award date:
2014

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2013–2014

Contribution à l'adoption des bases de données NoSQL

Martial LAMOCK

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Anthony CLEVE

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

Ce mémoire a pour objectif d'aider à répondre aux problématiques qui se posent lorsque l'on décide d'adopter la technologie NoSQL. Il existe plusieurs familles de bases de données NoSQL et il est parfois difficile de faire des choix face à ces diférentes plateformes.

Ce mémoire commencera par introduire les concepts liés aux bases de données en général et par la suite plus particulièrement aux bases de données NoSQL. Il fera un état de l'art de la technologie NoSQL en y décrivant les cas d'utilisation les plus appropriés à chaque famille de systèmes NoSQL. Ensuite il identifiera et analysera trois scénarios d'adoption de la technologie NoSQL. Enfin, il apportera une contribution à cette adoption en proposant un outil capable de réaliser automatiquement une partie de la migration des données d'une base de données relationnelle vers une base de données NoSQL.

Remerciements

Je remercie sincèrement mon promoteur et professeur Anthony Cleve de m'avoir soutenu dans l'élaboration de ce travail, pour ses bons conseils et sa patience.

Je remercie également mes autres professeurs pour leurs cours passionnants ainsi que mes camarades de classe.

Pour terminer, un merci tout particulier à Bilou, pour m'avoir supporté pendant toutes ces années.

Table des matières

Table des figures

Introduction

Les bases de données relationnelles se sont imposées aussi bien dans les entreprises que pour un usage plus personnel. Ces bases de données sont développées et optimisées depuis plusieurs dizaines d'années à tel point qu'elles offrent une réelle solution à différents types de problèmes de stockage. Ces bases de données relationnelles sont, de plus, devenues populaires avec l'arrivée du web et sont devenues des standards.

Mais le nombre d'accès à des données distantes ainsi que le nombre d'opérations demandées sur ces bases de données n'a fait qu'augmenter avec la croissance, toujours exponentielle, de l'utilisation du web. Dans un contexte poussé par le volume de données exploitées par les géants de l'Internet tel que Google, Facebook et Twitter, les bases de données dites classiques ne suffisent plus. Facebook, à lui seul, proposent des données à plusieurs millions de personnes potentiellement au même moment.

Outre la croissance du volume de données à traiter, il y a aussi un aspect de performance exigé par les utilisateurs. Les bases de données se doivent de répondre aux requêtes aussi vite que possible. C'est dans ce contexte de BigData que les bases de données NoSQL sont apparues et sont devenues plus populaires.

Néanmoins le passage à ce type de base de données n'est pas sans conséquences et soulève de nouveaux problèmes.

Nous pouvons citer le manque de formation à ces nouvelles technologies, certainement dû au manque d'experts en la matière. Ensuite, nous pouvons également citer le peu de standardisation qui existe entre ces plateformes, notamment au niveau des langages de requêtes.

Enfin, concernant l'adoption en tant que tel, un certain nombres de défis se posent également. Citons les trop rares guidelines et autres méthodologies

qui pourraient aider à l'adoption de ces technologies, notamment concernant le type de base de données NoSQL à choisir en fonction des cas d'utilisation. Cela pourrait aussi apporter une aide au processus de migration d'une base de données relationnelle, possédant un schéma de données, vers un modèle NoSQL, sans schéma.

Dans ce mémoire, nous tenterons d'apporter des réponses aux problèmatiques que nous venons d'évoquer.

Nous commencerons par faire un état de l'art des technologies NoSQL afin de pouvoir aider à faire le choix d'une famille technologique, en particulier, qui sera adaptée au mieux aux besoins que l'on a identifiés.

Nous analyserons ensuite trois scénarios d'adoption que nous avons identifiés. Ces trois scénarios sont, premièrement, une adoption *From Scratch* où nous ferons l'hypothèse d'un développement d'un nouveau système NoSQL ; deuxièmement, une adoption avec abandon total de SQL où nous ferons l'hypothèse d'une migration d'un système relationnel existant vers un nouveau système NoSQL et troisièmement, une adoption sans abandon de SQL qui correspond à faire coexister le système relationnel avec le système NoSQL.

Pour terminer, nous contribuerons modestement à cette problématique générale, en proposant un prototype permettant d'automatiser une partie des deux derniers scénarios d'adoption, à savoir la transformation du schéma d'une base de données relationnelle en NoSQL, ainsi que la migration des données du système relationnel vers une plateforme NoSQL.

Ce mémoire est structuré de la manière suivante ; le premier chapitre dresse un état de l'art des technologies NoSQL, ensuite, le chapitre deux identifie et analyse les trois scénarios d'adoption et le dernier chapitre présente notre contribution au développement d'un outil de migration.

Chapitre 1

Concepts de base

1.1 Base de données relationnelle

1.1.1 Modèle relationnel

Le modèle relationnel a été décrit pour la première fois par Edgar F. Codd dans son article *A Relational Model of Data for Large Shared Data Banks* [MB11]. Les bases de données, qui se basent sur le modèle de données relationnel, peuvent être représentées par des ensembles de relations entre tables où chaque table possèdent un nom unique.

Ces tables sont composées de colonnes ou appelées attributs ainsi que de lignes ou enregistrements. Chaque colonne a un domaine associé qui spécifie le type des valeurs des données qui peuvent être contenues dans celle-ci. Nous pouvons aussi définir des contraintes sur ces données telles que le caractère obligatoire de la colonne (non nul), les contraintes d'unicité ou encore les contraintes référentielles.

Chaque ligne est, elle, déterminée par une ou plusieurs valeurs qui forment la clé primaire. Celle-ci permet de n'avoir qu'une seule copie des données et donc d'éviter les redondances. En plus de cela, la clé primaire sert aussi d'index de telle sorte que cela favorise un accès rapide aux lignes de la table en se basant sur une ou plusieurs valeurs contenues dans ses colonnes.

Pour représenter les relations entre les tables, il y a le concept de clé étrangère. Des tables contiennent des clés étrangères qui sont en fait des clés primaires d'autres tables. Cela permet d'avoir une intégrité référentielle entre tables, c'est-à-dire que les clés étrangères ne peuvent prendre comme valeurs que les valeurs des clés primaires des tables de référence.

Cependant, il reste que des données peuvent encore se trouver en plusieurs exemplaires dans la même table, il y a alors de la redondance interne. Dans ce cas, on dit que la table est non-normalisée comme nous pouvons le voir à la figure 1.1

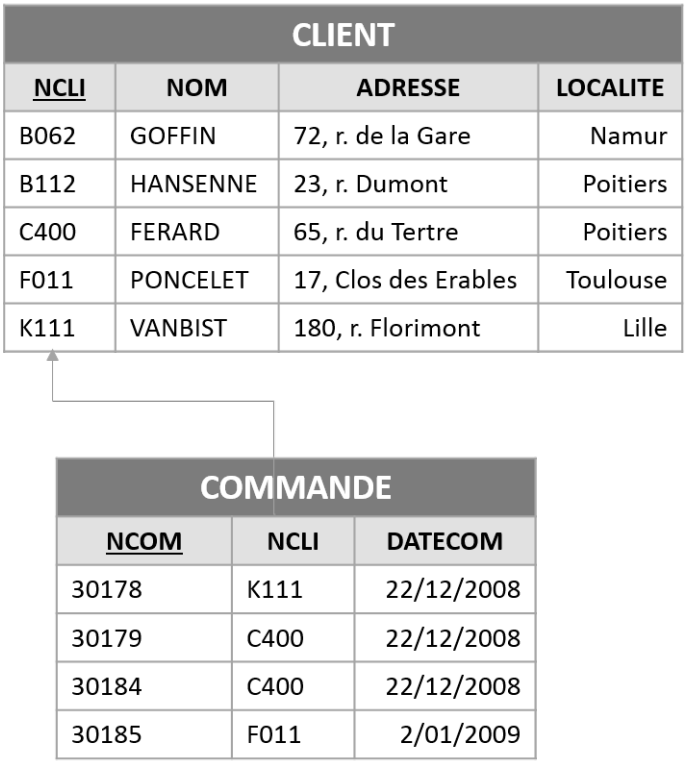
COMMANDE				
NCOM	**NCLI**	**NOM**	**ADRESSE**	**DATECOM**
30178	K111	VANBIST	180, r. Florimont	22/12/2008
30179	C400	FERARD	65, r. du Tertre	22/12/2008
30184	C400	FERARD	65, r. du Tertre	22/12/2008
30185	F011	PONCELET	17, Clos des Erables	2/01/2009

FIGURE 1.1 – Table non-normalisée

Afin d'éliminer le risque de redondance, on utilise le processus de normalisation qui va permettre de supprimer les données dupliquées en divisant la table concernée en fonction de ses clés primaires et dépendances fonctionnelles anormales. On dit qu'il y a dépendance fonctionnelle entre A et B notée $A \rightarrow B$ si connaissant une occurence de A on ne peut lui associer qu'une seule occurence de B. En d'autres termes, A détermine B. Par exemple, le numéro du client (NCLI) détermine le nom du client (NOM), noté $NCLI \rightarrow NOM$. Une dépendance fonctionnelle anormale quant à elle est une dépéndance dont le déterminant de la relation n'est pas identifiant [Hai12]. La normalisation permet de réduire la redondance et d'augmenter la maintenabilité et l'intégrité des données. Mais cela permet aussi de réduire la taille des tables et de les organiser en y mettant des contraintes.

Toutes les caractéristiques de la base de données relationnelle, comme ses tables, clés primaires, clés étrangères, ses contraintes, ses relations, etc. sont représentées selon ce qu'on appelle un schéma relationnel.

La figure 1.2 représente une modèle de données relationnel normalisé.



CLIENT			
NCLI	**NOM**	**ADRESSE**	**LOCALITE**
B062	GOFFIN	72, r. de la Gare	Namur
B112	HANSENNE	23, r. Dumont	Poitiers
C400	FERARD	65, r. du Tertre	Poitiers
F011	PONCELET	17, Clos des Erables	Toulouse
K111	VANBIST	180, r. Florimont	Lille

COMMANDE		
NCOM	**NCLI**	**DATECOM**
30178	K111	22/12/2008
30179	C400	22/12/2008
30184	C400	22/12/2008
30185	F011	2/01/2009

FIGURE 1.2 – Modèle relationnel classique

1.1.2 ACID

Une des principales caractéristiques des bases de données relationnelles est qu'elles ont un ensemble de propriétés qui assurent que les transactions se font de manières fiables. Une transaction est "une suite d'opérations constituant une unité logique de traitement et pour laquelle on exige globalement les garanties ACID" [Hai12]. Le paradigme de transaction est décrit dans l'article *Principles of Transaction-Oriented Database Recovery* [HR83] de T. Haerder et A. Reuter, d'où sont tirées les définitions qui vont suivre.

Ces propriétés sont l'atomicité (*Atomicity*), la cohérence (*Consistency*), l'isolation (*Isolation*) et la durabilité (*Durability*). Ce que l'on apelle les propriétés ACID.

En voici les définitions :

– *Atomicity* : La transaction doit être de type "$Tout - ou - rien$". Peu importe ce qu'il arrive, soit la transaction se déroule correctement, soit

celle-ci est annulée et les données reviennent à leur état de départ.

- *Consistency* : Losrqu'une transaction arrive correctement à sa fin, la cohérence des données doit être préservée. C'est-à-dire que l'on passe d'un état cohérent à un autre, et seul des modifications autorisées ont lieu.

- *Isolation* : Tous les événements qui se font à l'intérieur d'une transaction ne doivent être visibles de personne jusqu'à la fin de celle-ci. De plus, chaque transaction doit être indépendante l'une de l'autre.

- *Durability* : Une fois que la transaction est complète et que les données modifiées sont écrites dans la base de données alors on peut avoir la garantie que le résultat peut survivre à un dysfonctionnement.

1.2 Base de données NoSQL

Il ne suffit plus maintenant de n'avoir qu'une seule base de données où traditionnellement, les utilisateurs enregistraient leurs données dans des serveurs de bases de données spécialement dédiés à cela ou même sur leur propre disque dur. Les bases de données requierent souvent maintenant une rapidité d'accès, qu'elles soient distribuées et avec une grande disponibilité.

Un nouveau mouvement s'est développé début 2009 et a crû rapidement [Edl] dû à des géants du web tel que Google, Amazon, Facebook, etc. générant et devant gérer d'énormes quantités de données. Les bases de données relationnelles devenant dépassées ou non adaptées à ce genre d'utilisation.

Dès lors, nous avons vu apparaître ces nouvelles bases de données portant le nom de NoSQL. Ce terme fait penser à Non-SQL signifiant qu'il est opposé à SQL mais l'on y préfère la traduction Not Only SQL [Edl].

1.2.1 CAP

Les principales caractéristiques des bases de données distribuées en général ne sont pas les mêmes que vues précédemment. Ces propriétés sont la cohérence (*Consistency*), la disponibilité (*Available*) et la résistance au partitionnement ($Partition-tolerance$). Ces propriétés font partie du théorème CAP, aussi appelé le théorème de Brewer [GL02], et sont décrites ci-dessous.

– *Consistency* :
  La cohérence dans les propriétés ACID fait référence aux propriétés idéales que devraient avoir les données de la base de données lors de chaque transaction. Dans ce théorème CAP, le terme de cohérence signifie que la donnée à laquelle nous voulons accéder est bien la bonne et que la donnée retournée soit bien la dernière version de celle-ci. Par exemple, dans le cas de Facebook, cela serait que toutes les personnes qui désirent voir le statut de la même personne reçoivent bien le dernier statut de celui-ci et non un statut obsolète. Le but étant donc aussi que chaque personne voit au même moment la même donnée.

– *Available* :
  Cela signifie que le système doit rester disponible malgré l'arrêt de certains noeuds du système. C'est là qu'intervient la réplication des données afin d'être toujours certains d'y avoir accès. De même, pour l'exemple de Facebook, il ne faut pas qu'avec l'augmentation du nombre d'utilisateurs, ou la maintenance d'une partie du système, le site ne réponde plus. Toutes les demandes faites doivent pouvoir être traitées.

– *Partition − tolerance* :
  Cela signifie que s'il arrive qu'une partie du système distribué soit inaccessible, il soit toujours possible d'avoir accès à toutes les données. Il est logique que plus le système est répliqué, plus la résistance est grande et seule une panne générale du réseau peut alors empêcher l'accès au système.

Le théorème CAP dit qu'à un moment t, seules deux de ces propriétés peuvent être satisfaites en même temps. Cela veut dire que pour les systèmes distribués, il faudra faire un compromis et laisser une de ces propriétés de côté. Selon les choix qui peuvent être fait, nous obtiendrons des systèmes aux caractéristiques différentes.

Il existe donc trois cas différents, qui sont décrits ci-dessous :

– **Compromis C et A** :
  Ce cas signifie que pour autant que les noeuds soient disponibles, les données qui s'y trouvent seront cohérentes. On peut lire une même donnée depuis n'importe quel noeud, on aura le même résultat. En revanche, si un partitionnement se crée, il se peut que certaines données soient obsolètes. Plus il y aura de noeuds, moins le système sera disponible car il faudra de plus en plus de temps pour que tous les noeuds

soient synchronisés.

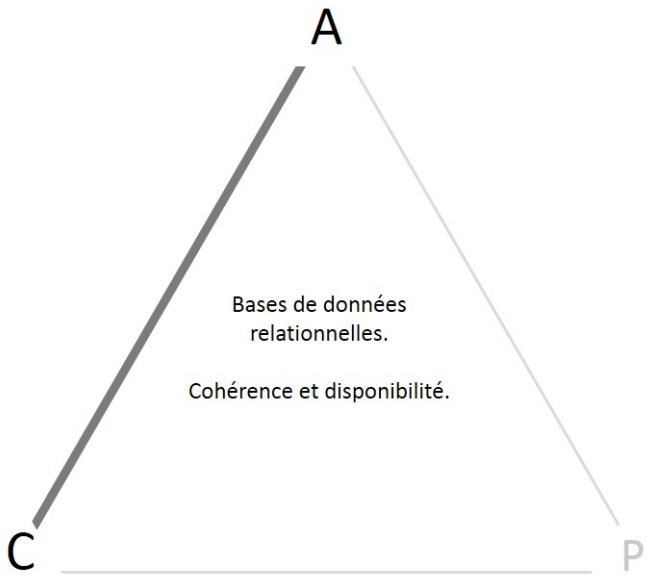


FIGURE 1.3 – Compromis C et A

– **Compromis A et P :**
Dans ce cas-ci, les données sont toujours accessibles, même dans le cas d'un partitionnement. Evidemment, il y aura des données qui ne seront pas cohérentes par rapport aux autres, mais cela peut être rétabli quand le problème de partitionnement est réglé. Dans le cas de données non critiques, c'est un compromis qui convient parfaitement. La propriété de cohérence est ici mise légerement de coté, mais cela ne veut pas dire que le système n'est jamais cohérent. Nous dirons donc que le système est *Eventually Consistent*. Ces caractéristiques sont, la plupart du temps, celles des systèmes NoSQL. Avoir plusieurs noeuds permet également de répartir la charge des requêtes. De plus, une haute disponibilité est demandée car l'utilisateur a généralement besoin d'une réponse rapide.

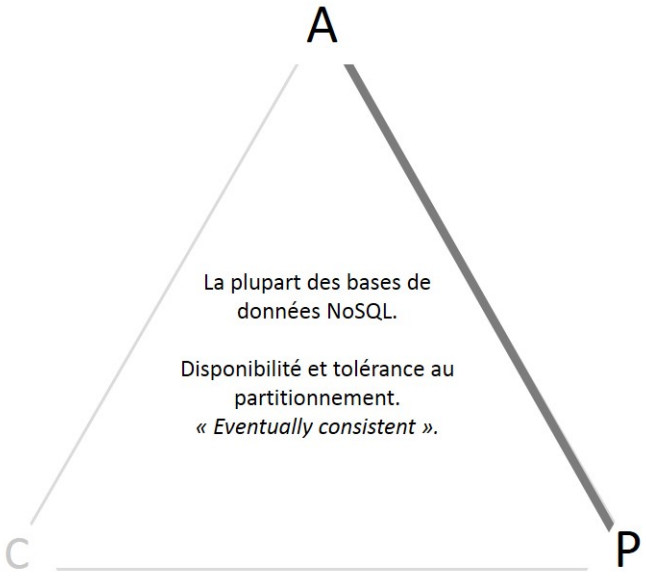


FIGURE 1.4 – Compromis A et P

– **Compromis C et P :**
Dans ce cas-ci, le système est hautement cohérent et résistant au partitionnement mais en revanche, il se peut que certaines données ne soient pas accessibles lorsqu'un noeud est en panne. Mais dès que la panne est rétablie, le système est de nouveau opérationnel. On retrouve ces caractéristiques dans certaines bases de données NoSQL, là où le système a besoin d'être très cohérent et pas seulement "eventually consistent".

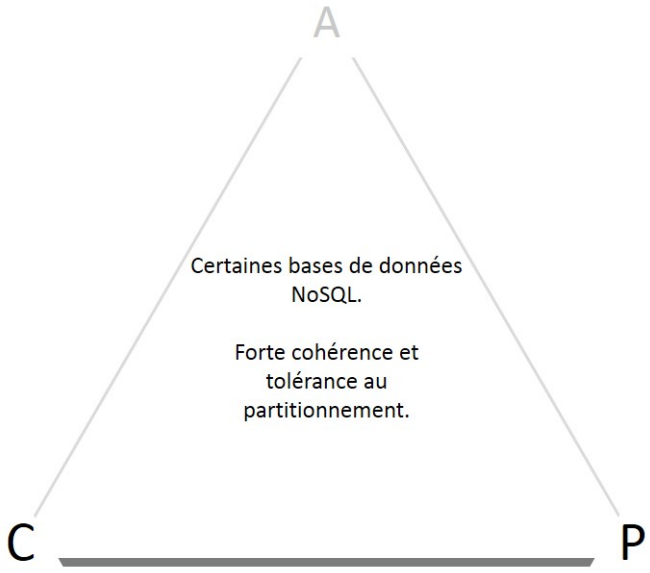


FIGURE 1.5 – Compromis C et P

1.3 Comparaisons

1.3.1 ACID vs BASE

Comme nous l'avons vu dans le théorème CAP, selon le type de système que nous voulons, certaines propriétés ne sont pas applicables. Notamment, nous avons vu que la plupart des bases de données NoSQL se permettent d'être moins strictes vis-à-vis de la cohérence et c'est ce que l'on appelle *Eventually Consistent*.

Avec ce type de base de données il n'est plus possible de promettre d'avoir les propriétés ACID comme les bases de données traditionnelles et pour faire face à cela une alternative à ACID est proposée [Pri08] et [Bro] . Il s'agit du modèle BASE qui, comme son nom le suggère, est l'opposé du modèle ACID. L'acronyme BASE signifie *Basically Available*, *Soft-state*, *Eventually Consistent*.

Les propriétés sont décrites ci-dessous :

– *Basically Available* :

Cela signifie, comme nous l'avons vu dans le théorème CAP, que les données seront toujours disponibles. Les requêtes atteindront toujours leur but, mais il se peut que, lors de la réponse, les données ne soient pas cohérentes ou que ces données retournées ne soient pas la dernière version de celles-ci.

- $Soft - state$:
  Cela signifie que l'état de la base de données n'est pas fixe et peut changer au cours du temps, même sans mise à jour par un utilisateur. Cela est dû à la propriété "eventually consistent", afin de rendre tous les noeuds cohérents.

- *Eventually Consistent* :
  Selon [Vog09], "eventually consistent" est une forme de cohérence faible. On dit qu'un système est "eventually consistent" s'il permet de garantir que s'il n'y a aucune mise à jour de la base de données, finalement tous les accès à la même donnée retourneront la dernière version de cette donnée. C'est-à-dire qu'à un certain moment, tous les noeuds finiront par présenter la même donnée. La période entre la mise à jour et le moment où tous les noeuds sont eux mis à jour s'appelle la fenêtre d'incohérence. Si l'on veut une grande disponibilité, cette propriété est importante car sans cette tolérance à l'incohérence, la modification d'une donnée devrait se propager sur tous les noeuds et forcer leur cohérence avant de pouvoir signifier que la modification a été correctement exécutée.

1.3.2 Autres foncionnalités

- **Systèmes répliqués/distribués** :
  Les nouvelles bases de données NoSQL mettent l'accent sur la disponibilité plutôt que sur la cohérence. Cela est d'autant plus simple que ces technologies permettent d'être distribuées et répliquées. Cet avantage a un coût lié au developpement d'applications car celles-ci doivent être capables de gérer plusieurs noeuds plutôt qu'un seul.

- **Scalabilité horizontale/verticale** :
  Avec l'avènement du BigData, une énorme quantité de données est brassée. Pour permettre la gestion de cette quantité croissante, les bases de données NoSQL permettent une croissance horizontale par exemple en divisant la requête en processus concurrents ou en ajoutant des

noeuds au système. Tandis que les bases de données classiques ne permettent généralement qu'une croissance verticale sur le même serveur par l'ajout de mémoire, de processeurs, etc. et cela a évidemment un coût supérieur [IJWIS2013a].

- **Evolutivité** : Les bases de données NoSQL ont l'énorme avantage d'être ce que l'on appelle "schema-less". C'est-à-dire sans schéma prédéfini comme peut l'être une base de données relationnelle. Cela veut dire qu'à tout moment, si les besoins métiers changent, il est toujours possible d'adapter le modèle, la structure des données.

- **Modèles de données** : Nous avons vu ci-dessus ce qu'était le modèle de données relationnel. Afin de manipuler des données qui sont difficilement représentables dans des tableaux, les bases de données NoSQL offrent différents modèles de données. Sur base de ces différents modèles, nous allons retrouver quatre types de bases de données NoSQL : *Key-Value*, *Document-oriented*, *Column-Family* et *Graph-oriented*. Comme le suggère [Sad13], les trois premiers types seront appelés des modèles de données aggrégés. Nous détaillerons ces quatres modèles dans la section suivante.

- **Language des requêtes** :
  Contrairement aux bases de données relationnelles, qui sont utilisées depuis plus de 20 ans et qui utilisent un langage commun qui est SQL, les bases de données NoSQL n'ont pas encore un langage standard. Il est difficile d'envisager un tel langage compte tenu des différences entre les modèles de données et le fait qu'il n'y ait pas de schéma pour ces données. Malgré cela, il existe des langages comme *UnQL* (*Unstructured Data Query Language*)[UnQ], qui est une sorte de langage SQL réalisé par CouchDB [Cou] et SQLite [SQL]. Aussi, [MB11] apportent une solution qui serait un langage de requête standardisé qui se base sur un modèle mathématique afin d'unifier SQL et NoSQL.

- **Rôle du développeur** :
  Le développeur a certainement un rôle plus important pour les bases de données NoSQL. Là où les propriétés ACID des bases de données relationnelles permettaient par exemple de n'avoir accès aux données que par un seul utilisateur, cela pourrait être au développeur d'implémenter lui-même cette fonctionnalité si la technologie ne la propose pas. C'est également le cas pour des structures de données complexes. Là où les bases de données relationnelles ont un schéma structuré et stable

pour ses données, une base de données NoSQL permet d'y stocker des données en tout genre et c'est également au développeur de gérer les différents types qu'il peut rencontrer.

1.4 Classification

Nous avons vu ci-dessus que nous retrouvions quatres modèles de données au sein des technologies NoSQL. Il s'agit des modèles *Key-Value*, *Column-family*, *Document-oriented* et *Graph-oriented*. Nous allons décrire chacun de ces modèles sur base de leur modèle et structure des données, leur modèle de cohérence et leurs avantages et inconvénients [MCC14] et [Sad13].

1.4.1 Key-Value

Le modèle *Key-Value* ou Clé-Valeur est aussi appelé *Key-Value Store* [Ria] [Dat]. C'est en fait le plus simple des modèles présentés ici. Il s'agit d'une simple table composée d'une colonne avec les clés (une chaine de caractère) et d'une colonne avec les valeurs. Chaque clé ne retroune qu'une et une seule valeur. Nous pouvons voir cela comme une table d'une base de données relationnelle où il n'y a que deux colonnes dont une (la clé) est clé primaire. Nous pouvons encore voir cela comme un dictionnaire où la clé est le mot et la valeur est la ou les définitions. La différence entre une table relationnelle a deux entrée et le modèle clé-valeur est que la valeur de ce dernier peut être de n'importe quelle nature.

Si nous voulons représenter le contenu de notre table relationnelle CLIENT-COMMANDE de la figure 1.2 avec le modèle clé-valeur, nous pourrions le faire de la manière montrée à la figure 1.6. Evidemment, cela n'a pas d'intêret car nous perdons toutes les relations et même la signification de ces valeurs.

Key	Value
001	B062
002	B112
003	C400
004	F011
005	K111
006	GOFFIN
007	HANSENNE

Key	Value
008	FERARD
009	PONCELET
010	VANBIST
011	30178
012	30179
013	30184
014	30185

FIGURE 1.6 – Key-Value Store - version naïve

Nous pouvons quand même envisager d'utiliser ce genre de modèle en profitant de la propriété qu'ont ces technologie de pouvoir stocker des valeurs dont le type peut être choisi. Nous proposons à la figure 1.7 de prendre comme valeurs des objets représentant une ligne de notre exemple de table relationnelle et dont la clé sera l'identifiant de la ligne. Nous pourrions même aller un peu plus loin en séparant les commandes des clients en les structurant dans des conteneur séparés appelés *Bucket* comme le montre la figure 1.8.

Key	Value
B062	« Nom : GOFFIN, ADRESSE: 72, r. de la Gare, LOCALITE: Namur »
B112	« Nom : HANSENNE, ADRESSE: 23, r. Dumont, LOCALITE: Poitiers »
C400	« Nom : FERARD, ADRESSE: 65, r. du Tertre, LOCALITE: Poitiers, COMMANDE: [30179, 30184]»
F011	« Nom : PONCELET, ADRESSE: 17, Clos des Erables, LOCALITE: Toulouse, COMMANDE: 30185 »
K111	« Nom : VANBIST, ADRESSE: 180, r. Florimont, LOCALITE: Lille, COMMANDE: 30178 »

Key	Value
K111	« Nom : VANBIST, ADRESSE: 180, r. Florimont, LOCALITE: Lille, COMMANDE: 30178 »
30178	« NCLI: K111, DATECOM: 22/12/2008 »
30179	« NCLI: C400, DATECOM: 22/12/2008 »
30184	« NCLI: C400, DATECOM: 22/12/2008 »
30185	« NCLI: F011, DATECOM: 2/01/2009 »

FIGURE 1.7 – Key-Value Store - élaborée

Les seules opérations que nous pouvons réaliser sur ce type de base de

données est la lecture (*get*), l'écriture (*put*) ou la suppression (*delete*). Dans le cas de l'écriture, si une valeur est déjà associée à la clé alors la valeur est tout simplement écrasée.

La figure 1.8 montre l'équivalent des éléments de la base de données relationnelle aux éléments de ce type de base de données NoSQL.

SQL	Key-Value
Database	Cluster
Table	Bucket
Row	Key value
rowid	Key

FIGURE 1.8 – Mapping RDBMS - Key-Value (Riak) [Sad13]

Structures des données

La structure des données qui sont acceptées dans ce type de base de données sont multiples. Cela peut être aussi bien des blob (*Binary Large OBject*, qui sont des fichiers binaires représentant aussi bien des images, des vidéos que des pages web), du texte, du JSON, etc. La figure suivante montre comment nous pouvons utiliser des url comme clé.

Modèle de cohérence

Lorsque la base de données de type *Key-Value* n'est pas distribuée, la cohérence des données peut être garantie, comme pour les bases de données relationnelles. Lorsque le système est distribué, nous obtenons les mêmes problèmes que vus précédemment avec le théorème CAP. C'est-à-dire que l'on obtient un modèle "eventually consistent".

Avantages

Ce type de base de données est intéressant pour par exemple y stocker des sessions web, où un simple *put* ou *get* permet de stocker ou d'aller chercher la session d'un utilisateur. Il est également utilisé pour des paniers e-commerce,

où toutes les informations sont stockées comme une seule valeur et où la clé est le *userid*.

Inconvénients

Comme nous l'avons vu, nous pouvons représenter le *Key-Value Store* comme un tableau à deux colonnes. De ce fait, il n'est pas judicieux de l'utiliser pour représenter des relations entre les différentes données, comme on peut le retrouver dans des bases de données relationnelles.

1.4.2 Column-family

Ce type de base de données fonctionne un peu de la même manière que le type *Key-Value Store*. Les clés sont liées à des valeurs, mais dans ce type, les valeurs seront des familles de colonnes où chaque famille de colonnes contient un certain nombre de données. Comme nous pouvons le voir dans la figure 1.9, à chaque ligne, identifiée par sa Row key, correspond un certain nombre d'informations. Contrairement à une base de données relationnelle, le nombre de colonnes ne doit pas être le même pour toutes les lignes.

Il est aussi intéressant de noter qu'à chaque valeur est associée un *horodatage* ou *timestamp* en anglais, qui est en fait une date et une heure où la valeur a été mise à jour pour la dernière fois. Ce mécanisme permet de gérer les conflits d'écriture, de faire expirer une valeur, etc.

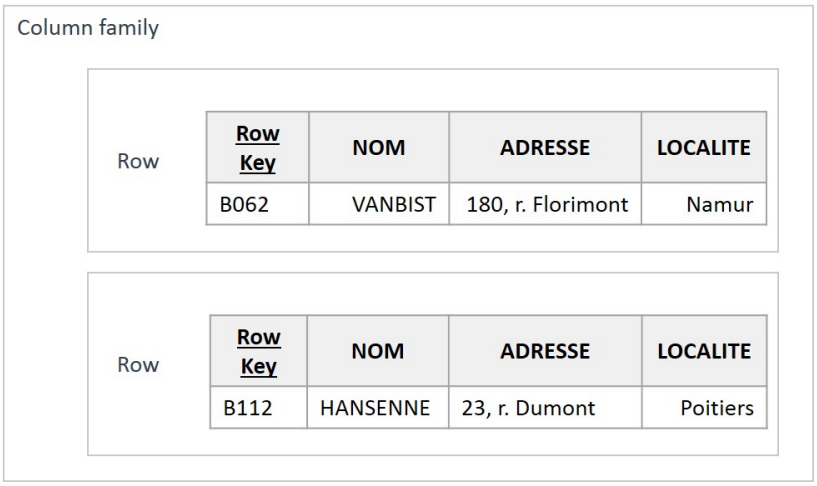


FIGURE 1.9 – Column-Family [Sad13]

La figure 1.10 montre l'équivalent des éléments de la base de données relationnelle aux éléments de ce type de base de données NoSQL.

SQL	Column-family
Database	Keyspace
Table	Column-family
Row	Row
Colonne	Colonne

FIGURE 1.10 – Mapping RDBMS - Column-Family (Cassandra) [Sad13]

Structures des données

Etant basé sur le type *Key-Value Store*, le type *Column-family* accepte les même structures de données.

Modèle de cohérence

Nous allons prendre le cas de Cassandra [Cas] pour illustrer les différents niveau de cohérence qu'il est possible d'avoir. Premièrement, il est possible de demander le nombre de réplications de la donnée que nous souhaitons

avoir. Cela permet de s'assurer une certaine fiabilité des données. Ensuite, il est possible de demander que le système, lors d'une lecture, ne donne le résultat que du premier réplicat. Lorsque la donnée est indisponible ou perdue, on passe au deuxième réplicat, et ainsi de suite. Cela permet une disponibilité plus grande mais en dépit de la cohérence. En revanche, il est possible de demander au système, lors d'une lecture, de vérifier la valeur de tous les noeuds. Si les valeurs ne sont pas cohérentes, l'utilisateur en sera informé. Le choix de la cohérence dépend de l'importance et la criticité des données.

Avantages

Ce type de base de données est pratique lorsque l'utilisateur a besoin de stocker des informations en rapport à des évenements. Le timestamp permet de retrouver l'heure de l'événement qui sera la clé, tandis que les informations de l'événement peuvent être facilement stockées sous forme de colonnes. Ce type est également utilisé pour y stocker des colonnes qui peuvent se supprimer après un certain temps. Lorsque ce temps (TTL pour Time To Live) est dépassé, la colonne et ses valeurs sont supprimées.

Inconvénients

De par la structure de ces lignes, avec le nombre de colonnes qui peut fluctuer, il est par exemple difficile de calculer une somme pour une certaine colonne etc. Il est nécessaire dans ce cas, que ce soit le client qui fasse ces opérations, après avoir rapatrié les données.

1.4.3 Document-oriented

Ce type de modèle possède également quelques similitudes avec le modèle *Key-Value Store*. L'inconvénient qu'avait ce dernier était qu'il est impossible de faire une recherche sur base de la valeur de la paire clé-valeur. Le type orienté-document permet justement de pallier à ce problème. Ici, lorsqu'un document est inséré dans la base de données, tout son contenu est directement indexé, de telle manière qu'il est possible d'interroger la base de données sur n'importe quelle valeur.

Ce type de base de données a également des similitudes avec les bases de données relationnelles. Comme nous pouvons le voir dans la figure 1.11,

chaque document sera classé dans des collections qui correspondent à des tables dans les bases de données relationnelles. Mais ce qui est intéressant de remarquer est que le langage de requête de ces bases de données orienté-document est presque aussi complet que le langage SQL étant donné que toutes les valeurs des documents sont indexées et interrogeables, ce qui est par exemple le cas de MongoDB [Mon].

SQL	MongoDB
database	database
table	collection
row	document
colonne	champs
clé primaire	champs _id

FIGURE 1.11 – Mapping RDBMS - Document-Oriented (MongoDB) [Mon]

La figure 1.12 montre une partie de notre exemple CLIENT-COMMANDE en utilisant le modèle de données de MongoDB.

FIGURE 1.12 – Modèle de données Document-oriented

Structures des données

Il faut voir la structure de ce modèle de données comme une structure d'arbre, comme montré à la figure 1.13. La base de données contient des documents rangés dans des collections. Chaque document peut également contenir d'autres documents. Nous pouvons voir cela comme le système de

dossiers dans Windows.

De même que pour le type de base de données *Column-Family*, il n'est pas nécessaire d'avoir les mêmes nombres d'attributs dans les différents documents.

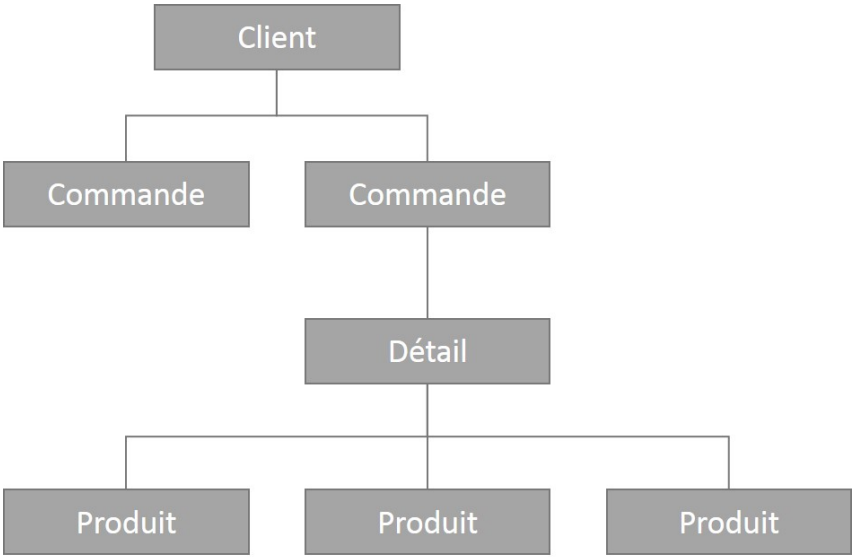


FIGURE 1.13 – structure d'arbre du Document-Oriented [Sad13]

Modèle de cohérence

Pour ce cas-ci, nous allons nous baser sur le modèle de cohérence du stystème MongoDB [Mon]. Celui-ci permet d'avoir des réplicats pour assurer une disponibilité et fiabilité, mais MongoDB propose également par exemple, lors de l'écriture, de définir sur combien de réplicats il faut écrire avant de notifier la réussite de cette écriture. Si cela a évidemment un impact plutôt négatif sur la performance, l'impact sur la cohérence est lui plutôt positif. En effet, si il est autorisé de lire une donnée sur n'importe quel réplicat, il se peut que la valeur ne soit pas cohérente. De ce fait, MongoDB permet à l'utilisateur de choisir entre cohérence totale ou "eventually consistent". Comme nous l'avons déjà remarqué ci-dessus, cela dépendra de la criticité des données.

Avantages

Avec ce genre de base de données, il est simple de retrouver par exemple, tous les documents contenant une certaine valeur, contenant certains mots, etc. Il est également utilisé pour les blogs, où les articles, commentaires, informations sur les auteurs sont tout à fait structurés pour correspondre à des documents, même imbriqués.

Inconvénients

Nous serons d'accord de dire que ce type de base de données n'est pas optimisé pour la gestion de données fortement liées entre elles comme pourrait l'être une base de données relationnelle. Par exemple, il sera généralement plus compliqué de réaliser des opérations sur plusieurs documents liés entre eux.

1.4.4 Graph-oriented

Les bases de données orientées graphes sont spécialement conçues pour représenter des objets et les relations qui existent entre ceux-ci [Neo]. Elles contiennent des noeuds, qui sont des instances de ces objets et contiennent également des arcs entre ces noeuds et représentent des relations. Aussi bien les noeuds que les arcs peuvent être d'un certain type et avoir des propriétés qui leur sont associées. Enfin, les arcs peuvent aussi être orientés ou non afin de dénoter une certaine relation. La figure 1.14 en est l'illustration. L'ensemble de ces noeuds et leurs relations forment un graphe. Ces bases de données sont intéressantes pour représenter des problèmes complexes que nous pouvons retrouver dans les réseaux sociaux.

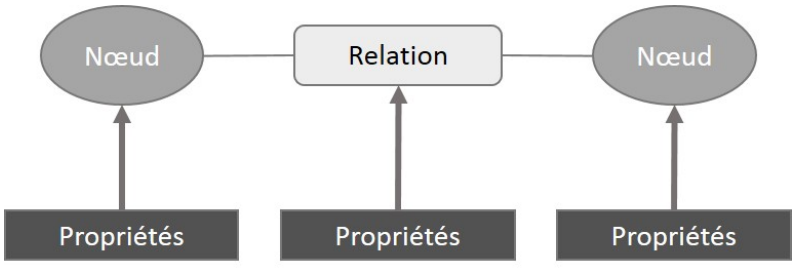


FIGURE 1.14 – Noeuds, relations et propriétés pour le type Graph-Oriented [Sad13]

Structures des données

Nous allons cette fois-ci nous baser, non pas sur l'entiereté de notre exemple CLIENT-COMMANDE mais bien sur les clients comme nous pouvons le voir à la figure 1.15. Nous remarquons que les types d'arcs organisent d'une certaine manière les différents noeuds. Les noeuds tout comme les arcs peuvent contenir des propriétés. Les noeuds B062, B112 et C400 contiennent les propriétés "Nom" et "Sexe". Le type d'arc entre les noeuds B062 et B112 est "Marié" et cette relation ne doit pas être orientée, ou plutôt elle concerne les deux noeuds. Nous pouvons lui assigner une propriété, par exemple "Date", qui est la date de leur mariage. En revanche, nous remarquons que les arcs qui vont de B062 à C400 et B112 à C400 sont de type "Parent de" et celles-ci doivent être orientées.

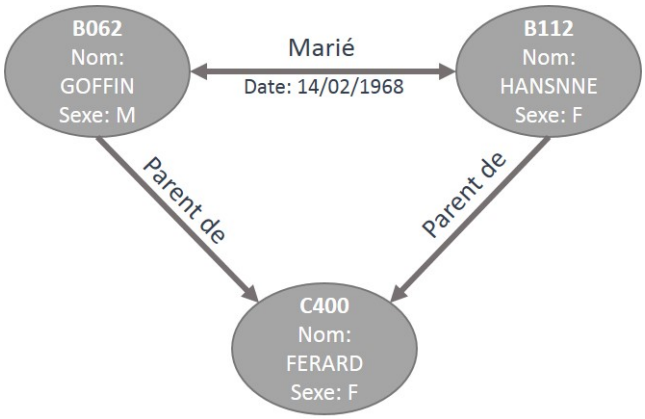


FIGURE 1.15 – Graph-Oriented [Sad13]

Modèle de cohérence

La plupart des bases de données orientées graphes ne supportent pas la distribution. Lorsque la base de donnée se trouve sur un seul serveur, la cohérence peut être assurée. Lorsque plusieurs serveurs entrent en jeu, due à la complexité des relations entre noeuds, il est difficile d'assurer la cohérence entre les données.

Avantages

Les avantages d'utiliser une telle base de données plutôt qu'une base de données relationnelle sont multiples lorsqu'il s'agit de représenter des re-

lations. Si nous prenons l'exemple d'une base de données relationnelle qui stocke une liste de frères et soeurs. Il est simple de demander par exemple le nom de toutes les personnes qui sont soeurs. Lorsque nous voulons rajouter une nouvelle relation, il est souvent nécessaire de repenser le schéma de la base de données. Tandis que pour une base de données orientée graphe, il suffit de rajouter un arc entre les noeuds.

Un autre exemple peut être montré en prenant le cas de LinkedIn, le réseau social professionnel bien connu. Dans une base de données relationnelle classique, il est difficile de représenter à quel degré nous sommes en lien avec une autre personne. Avec les graphes, des règles peuvent être imposées comme c'est le cas dans ce réseau social où il n'est pas possible de se connecter directement avec une personne dont le lien est de degré 3. C'est-à-dire l'ami des amis des amis.

Il est aussi possible de calculer quel est le chemin le plus court pour passer d'un noeud à une autre et de se baser sur des propriétés spécifiques ; choses qui sont plus difficiles dans les bases de données relationnelles.

Inconvénients

Comme nous l'avons vu dans ses avantages ci-dessus, les bases de données orientées graphes sont utiles quand nous devons représenter un système avec des relations complexes. Sur ce principe, il n'est pas approprié de représenter des structures de données complexes comme des documents imbriqués sous forme de graphe. De la même manière, ce type de base de données a plus une utilité pour l'analyse que pour le stockage. De plus, si nous prenons le cas de Neo4J [Neo] il n'est pas possible de représenter des objets ayant des relations sur eux-mêmes.

Chapitre 2

Adoption

Nous avons vu dans le chapitre précédent les différences qu'il pouvait y avoir entre d'une part, les bases de données relationnelles et les bases de données NoSQL, mais aussi d'autre part, parmi les techologies NoSQL. Cela peut contribuer à aider les personnes désireuses de choisir une technologie NoSQL.

Ce chapitre 3 a lui pour objectif d'identifier les différents scénarios d'adoption de la technologie NoSQL choisie. Il identifie et analyse également les défis que ces différents scénarios posent.

Nous allons analyser trois scénarios d'adoption de la technologie NoSQL. Le premier est un scénario appelé From Scratch (section 2.1) qui ne se basera pas sur une base de données existante mais au départ de l'analyse des besoins métiers. Le deuxième scénario est une migration avec abandon total de SQL (section 2.2), où l'on partira d'une base de données relationnelle existante pour cette migration. Enfin, le troisième scénario d'adoption sera une migration sans abondon de SQL (section 2.3). Il en résultera une coexistence partielle ou totale entre SQL et NoSQL.

Dans la suite de ce chapitre, nous avons choisi d'illustrer les propositions avec une technologie orientée document. Comme nous l'avons vu dans le chapitre précédent, cette technologie possède des points communs avec la technologie SQL. Nous pouvons rappeler notamment la gestion des données sous forme de documents qui permet un mapping quasiment équivalement aux tables relationnelles. Les fonctions de recherche qui permettent de parcourir les documents non seulement sur base des valeurs mais également sur base des champs de ceux-ci. Nous avons, en particuler, choisi pour exemple la technologie MongoDB [Mon].

2.1 From Scratch

Dans ce scénario, nous faisons l'hypothèse que l'on souhaite adopter la technologie NoSQL dans le cadre du développement d'un "nouveau" système, ou d'un sous-système assez indépendant des systèmes existants. Dans ce scénario, il s'agit donc de suivre une méthodologie de conception de base de données, partant des besoins, en passant par l'analyse de ceux-ci afin de produire des modèles de données, pour arriver à une structure physique représentant au mieux ces besoins de départ. Une telle méthodologie n'existe pas ou peu pour le développement de base de données NoSQL. C'est pourquoi nous nous inspirerons des méthodologies standards de conception de base de données relationnelles. Nous nous baserons notamment en grande partie de celle proposée par J-L Hainaut [Hai12].

Nous suivrons une série d'étapes qui permettront de passer de l'analyse des besoins au schéma physique de la base de données NoSQL.

La figure 2.1 montre les différentes étapes que l'on suivra afin d'arriver à ce résultat. Premièrement, l'analyse conceptuelle, deuxièmement la conception logique et enfin troisièmement la conception physique.

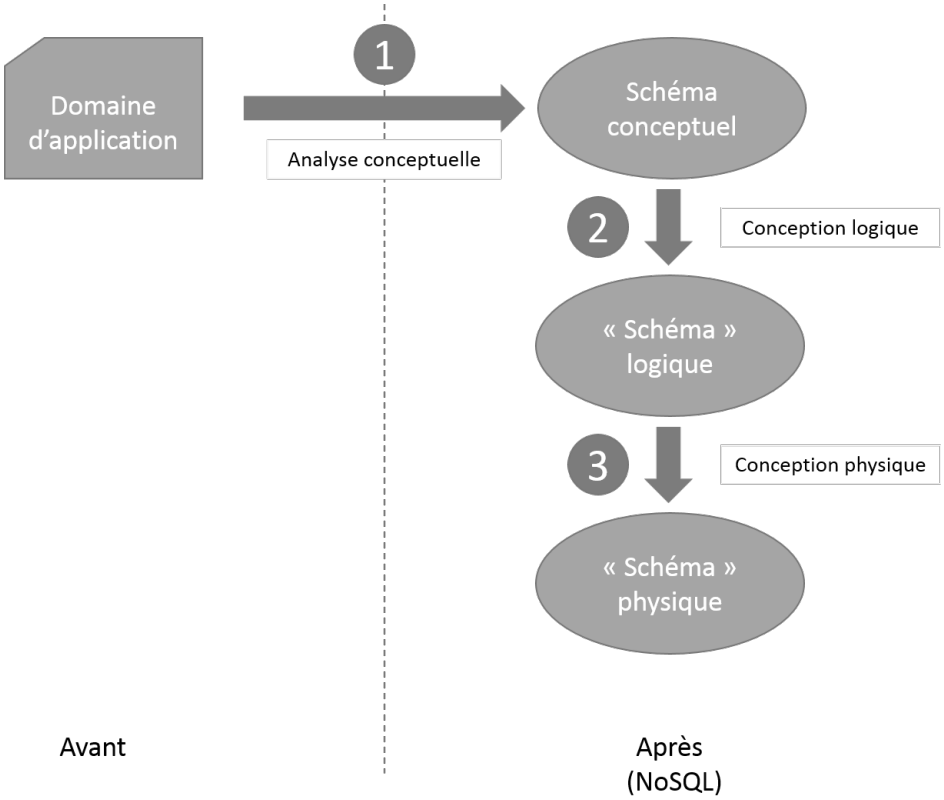


FIGURE 2.1 – Etapes du scénario From Scratch

2.1.1 Analyse conceptuelle du domaine d'application

Cette analyse a pour but la construction d'un schéma conceptuel sur base du domaine d'application. Ce genre d'analyse est bien connu et exploite le modèle entité-association [Hai12].

Cette étape est importante car elle modélise des situations réelles sur base, généralement, de texte, d'interviews ou encore d'observations. Nous ne détaillerons le fonctionnement de l'analyse conceptuelle car il s'agit d'une méthodologie courante. Néanmoins, nous pouvons rappeler les principes de base du résultat de l'analyse qu'est le schéma conceptuel.

Modèle Entité-Association

Pour représenter le schéma conceptuel, c'est le modèle Entité-Association qui sera utilisé. Celui-ci est composé d'ensembles d'entités, dotées de pro-

priétés et en association les unes avec les autres.[Hai12]

Un type d'entité représente une entité du domaine d'application. Dans notre exemple habituel, il s'agit d'un CLIENT, d'une COMMANDE, etc. que l'on représente comme indiqué à la figure 2.2.

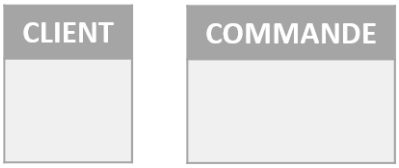


FIGURE 2.2 – Types d'entités

Chaque type d'entité possède des propriétés que l'on appelle attributs. Par exemple, chaque client est caractérisé par un numéro, son nom, son adresse, etc. Nous ajoutons donc au type d'entité CLIENT des attributs NCLI, NOM, ADRESSE, etc. comme le montre la figure 2.3.

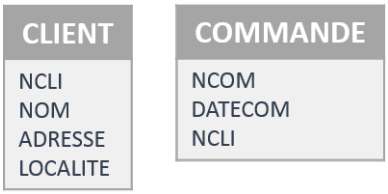


FIGURE 2.3 – Types d'entités et attributs

Ensuite, afin de correspondre à la réalité, il faut associer ces types d'entités ensemble. Cela se représente au moyen de type d'associations. Nous pouvons donc définir le type d'association *passe* entre CLIENT et COMMANDE, ce qui représente bien que le client passe une commande. Nous pouvons voir ce type d'association à la figure 2.4

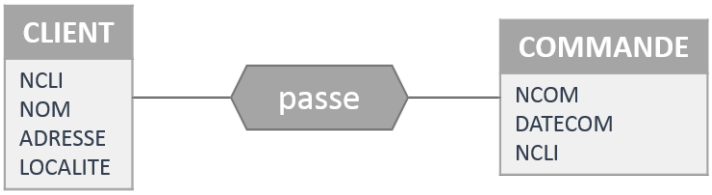


FIGURE 2.4 – Types d'entités, attributs et type d'association

Ensuite, à la représentation du modèle Entité-Association, nous devons aussi ajouter deux propriétés au type d'association. La première est la classe

fonctionnelle du type d'association et la deuxième est son caractère obligatoire ou facultatif. Cette première propriété décrit le nombre maximum d'entité A pour chaque entité B [Hai12] et vice-versa. Dans notre exemple, il s'agira de représenter le nombre de commandes que le client peut passer ainsi que le nombre de clients qui peuvent passer la commande. On appelle aussi cela la cardinalité de l'association. Comme le montre la figure 2.5, un client peut passer un nombre compris entre 0 ou N commande et une et une seule commande ne peut être passée que par un et un seul client.

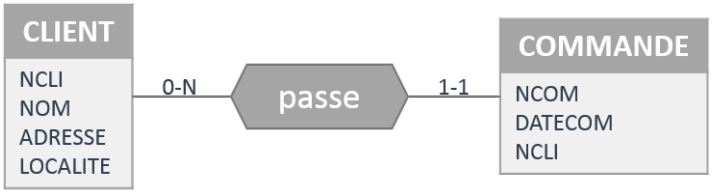


FIGURE 2.5 – Types d'entités, attributs et type d'association avec ses cardinalités

La deuxième propriété est donc le caractère obligatoire ou facultatif d'un type d'association. Ainsi dans notre exemple CLIENT-COMMANDE, il est obligatoire que l'entité COMMANDE participe à une association *passe* pour qu'elle existe. Si nous avions choisi de mettre 0-1 du coté COMMANDE, cela aurait signifié qu'une commande peut être passée par zéro ou un client, ce qui n'est pas possible dans la réalite. Nous ne nous attarderons pas sur ce caractère obligatoire ou facultatif.

Il existe donc trois classes fonctionnelles de type d'associations : *un-à-plusieurs*, *un-à-un* et *plusieurs-à-plusieurs* dont nous allons présenter brièvement un exemple ci-dessous.

un-à-plusieurs : Représenté par **1-N** ou **0-N**. La figure 2.5 est un exemple de ce type d'association.

un-à-un : Représenté par **1-1**. Un exemple serait qu'un client ne pourrait avoir qu'une seule adresse et une adresse ne peut être prise que par un seul client, comme le montre la figure 2.6.

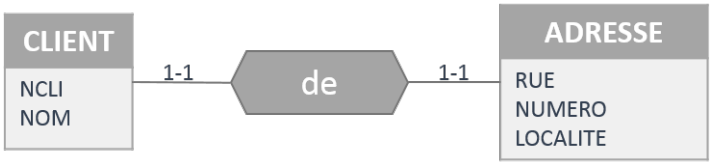


FIGURE 2.6 – Types d'association *un à un*

plusieurs-à-plusieurs : Représenté par **N-N** ou **0-N**. Un exemple serait qu'une usine peut frabriquer plusieurs produits et un même produit peut être fabriqué par plusieurs usines, comme le montre la figure 2.7.

FIGURE 2.7 – Types d'association *plusieurs à plusieurs*

Pour terminer, nous devons introduire le concept d'identifiant des types d'entités. C'est en fait un attribut qui permet d'identifier toutes les entités de ce type. Nous prendrons l'hypothèse que chaque type d'entité possédera un identifiant. Dans le modèle Entité-Association, cet identifiant sera l'attribut qui est souligné et sera aussi représenté par une clause $id : identifiant$, comme nous pouvons le voir dans la figure 2.8.

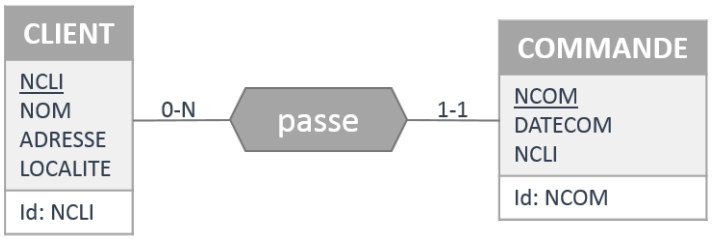


FIGURE 2.8 – Types d'entités, attributs, type d'association avec ses cardinalités et identifiants

Maintenant que nous avons obtenu notre schéma conceptuel sur base du domaine d'application ainsi que décrit de quoi était composé ce schéma, nous pouvons passer à l'étape suivante qui est la conception logique.

2.1.2 Conception logique

Nous sommes à la deuxième étape de notre scnéario From Scratch. Cette étape a pour but de produire un schéma [1] adapté au modèle de données de la famille technologique NoSQL. Donc le schéma logique que nous obtiendrons

1. Par facilité, nous accepterons ici cet abus de langage, NoSQL étant schemaless nous devrions plutôt parler de pseudo-schema.

sera totalement dépendant de la technologie choisie. Pour rappel, ce que nous définirons ci-après sera illustré pour MongoDB.

Comme nous l'avons vu dans le chapitre consacré aux concepts, les bases de données NoSQL orientées document proposent un mapping pour faire correspondre les tables, lignes, colonnes, etc. des bases de données relationnelles à des éléments spécifiques. De plus, nous avons aussi a disposition les règles de transformations du schéma conceptuel, c'est-à-dire le modèle Entité-Association, en schéma logique relationnel [Hai12].

Sur base de ce mapping et de ces règles de transformations, nous allons proposer nous aussi des règles afin de passer du schéma conceptuel, obtenu ci-avant, à un schéma logique adapté à notre technologie NoSQL.

Nous pouvons commencer à établir notre première règle qui est la transformation des types d'entité simple de notre schéma conceptuel.

Transformation de type d'entités

Tout d'abord, chaque type d'entités sera représenté par une collection qui portera le nom du type d'entité. Chaque entité du même type sera lui décrit comme un document. Enfin, chaque attribut présent dans les types d'entités sera représenté par un champ portant également le même nom que l'attribut. La figure 2.9 montre cette première règle de transformation. MongoDB étant Schemaless comme nous l'avons déjà expliqué, il n'est pas possible de créé un "schéma" sans valeur. Nous noterons donc ces valeurs $V_1 ... V_n$, mais celle-ci ne représenteront pas des valeurs réelles à proprement parler.

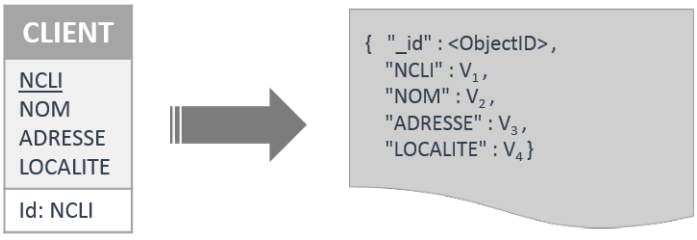


FIGURE 2.9 – Première règle de transformation

Notons que dès la première règle de transformation, nous sommes déjà amené à faire un choix concernant la représentation des attributs identi-

fiants dans le futur document. Comme nous pouvons le voir à la figure 2.9, un champ *_id* est présent. Celui-ci correspond à un champ identifiant pour MongoDB.

Nous avons donc deux possibilités de représentation. La première est de laisser MongoDB proposer automatiquement un identifiant, ce qui est le cas pour la figure 2.9. La deuxième solution est de spécifier un identifiant naturel, qui sera celui du type d'entité, comme montré à la figure 2.10. Nous privilégierons cette dernière solution car elle représente mieux le caractère identifiant de l'attribut.

FIGURE 2.10 – Première règle de transformation avec identifiant naturel

Nous avons vu la première règle qui consiste à transformer des types d'entités avec leurs attributs et leur identifiant. Nous avons vu auparavant qu'il existait trois classes fonctionnelles de types d'associations : *un-à-plusieurs*, *un-à-un* et *plusieurs-à-plusieurs*. Nous allons proposer une règle pour chacun de ces types d'associations.

De la même manière qu'il existe des règles de transformations des types d'associations pour obtenir un schéma logique relationnel [Hai12], nous allons nous baser sur celles-ci pour proposer à nouveau des règles adaptées à notre technologie.

Transformation du type d'associations un-à-plusieurs

Soit notre exemple CLIENT-COMMANDE ayant un type d'associations un-à-plusieurs comme le montre la figure 2.8. Pour se baser sur les règles de transformations en schéma logique relationnel, nous obtiendrons le résultat suivant à la figure 2.11.

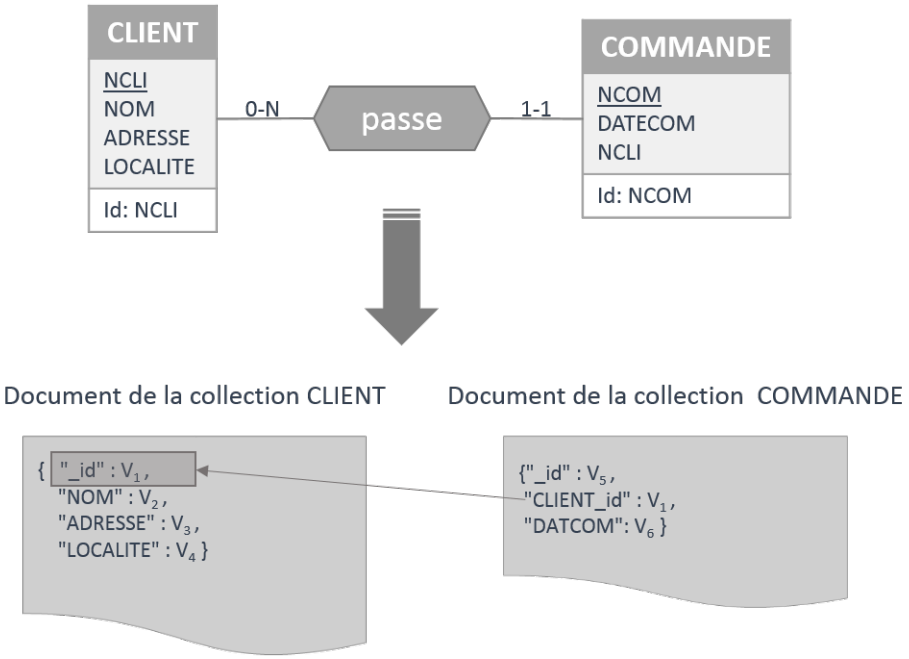


FIGURE 2.11 – Règle de transformation *un-à-plusieurs* par références

Nous pouvons tout d'abord observer que la première a bien été respectée. Ensuite, il suffit de référencer l'identifiant du type d'entités CLIENT dans le document COMMANDE, dont le nom de ce champ sera composé du nom du type d'entité de l'identifiant suivi, par pure convention, de "_id". Ce qui donne le champ "CLIENT_id".

La transformation que nous venons de voir est une approche normalisée au sens que nous avons vu dans le chapitre des concepts. Le document COMMANDE contient la référence du document CLIENT. Les informations sont contenues dans deux documents bien distincts.

Il existe certainement plusieurs méthodes de transformations et nous allons proposer une autre approche, qui celle-ci est dénormalisée. Parfois, pour des questions de performance, il est intéressant de rassembler les informations dans un même document. Notamment selon ce que l'on a identifié lors de l'analyse des besoins en amont du schéma conceptuel. Concrètement, pour retrouver deux documents comme CLIENT et COMMANDE, il faudra deux requêtes. Tandis que si ces deux documents n'en forment plus qu'un, une seule requête suffit.

Nous pouvons dire que NoSQL n'est plus seulement guidé sur base des

données comme l'est le modèle relationnel. Il est aussi guidé par les objectifs de l'adoption de cette technologie.

La deuxième approche serait de faire contenir les données concernant les commandes dans le document CLIENT, comme le montre la figure 2.12. Cette approche est appelée *Embedded*, qui signifie embarqué/imbriqué.

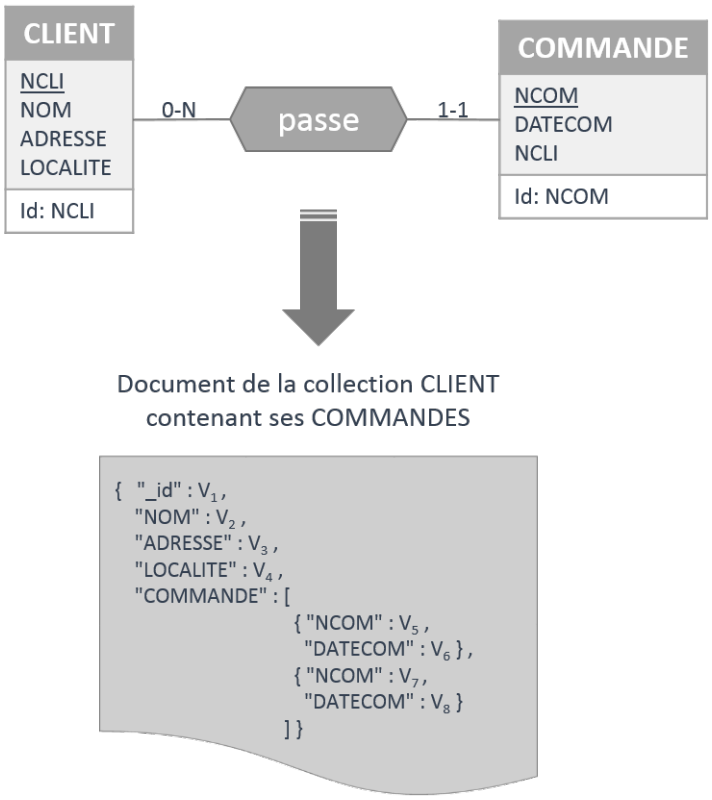


FIGURE 2.12 – Règle de transformation textitun-à-plusieurs *Embedded*

Nous pouvons formaliser cette approche en remarquant que le type d'entité CLIENT suit bien la première règle de transformation. Cependant, nous lui ajoutons un champ correspondant au nom du type d'entité COMMANDE. Ce champ sera composé d'un autre document contenant les attributs du type d'entité COMMANDE. Notons qu'il n'est plus nécessaire de spécifier l'attribut identifant de CLIENT dans le document COMMANDE lorsqu'il s'agit de l'approche *Embedded*.

A l'avenir, nous présenterons à chaque fois, si cela est nécessaire les deux approches qui sont les approches par référencement ou *Embedded*.

Passons maintenant à la transformation du type d'associations *un-à-un*.

Transformation du type d'associations un-à-un

Pour ce type d'associations, nous allons reprendre l'exemple de la figure 2.6 auquel nous avons rajouté les identifiants nécessaires et qui applique qu'un client ne peut avoir qu'une seule adresse et vice-versa. La figure 2.13 montre cet exemple complété ainsi que le résultat de l'application de la règle de transformation des types d'association *un-à-un*.

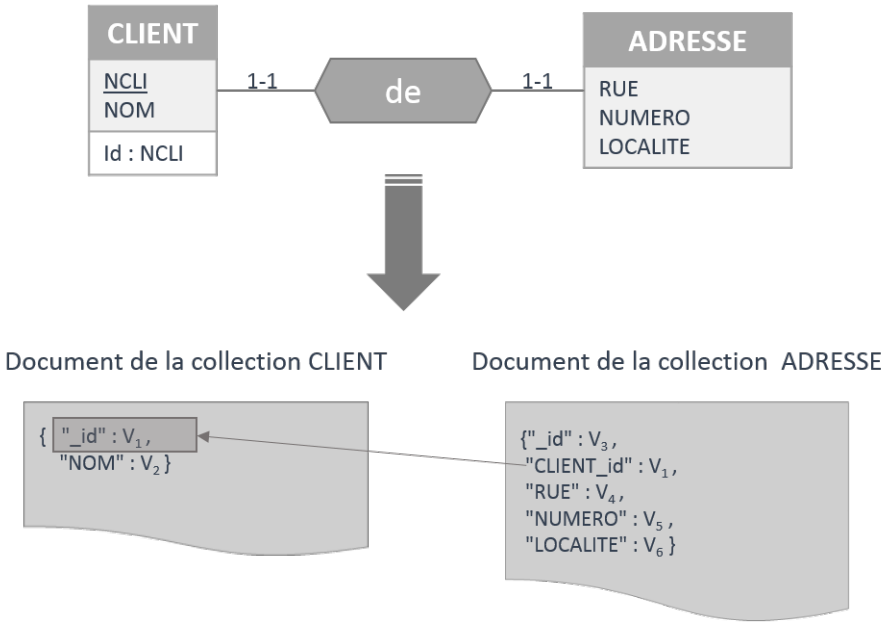


FIGURE 2.13 – Règle de transformation *un-à-un* par références

Selon le même principe que pour le type d'associations précédent, la première règle s'applique pour les deux types d'entités, mais le document venant du type d'entité ADRESSE référence l'"_id" du document CLIENT qui est l'attribut identifiant qui provient de son type d'entité. Notons que nous aurions pu donner au document COMMANDE le même identifiant que le document CLIENT, mais cela transgresse les règles établies par MongoDB qui veut que chaque "_id" soit unique.

Nous pouvons ici aussi proposer la deuxième approche qui est celle *Embedded*, comme nous le montre la figure 2.14.

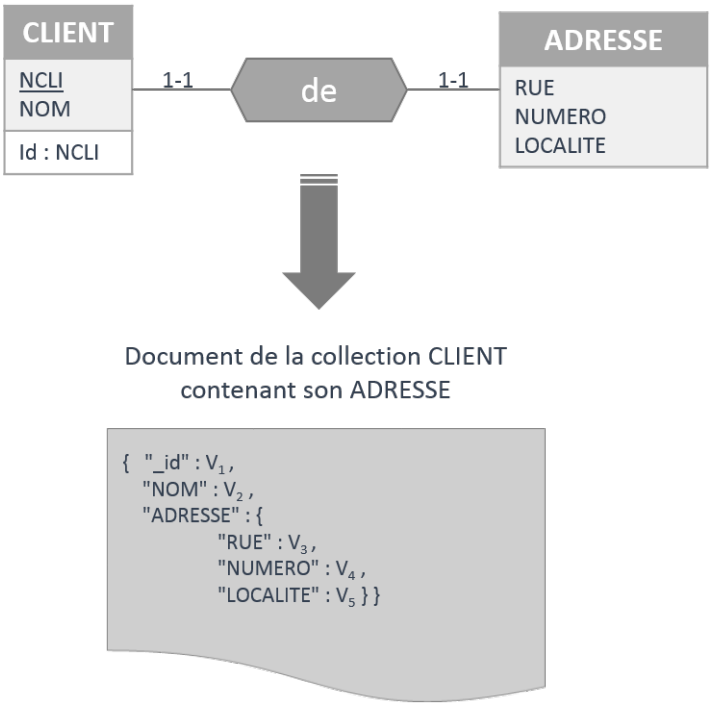


FIGURE 2.14 – Règle de transformation *un-à-un Embedded*

Dans ce cas-ci, les règles de transformation sont totalement identiques à la règle employée pour la transformation *un-à-plusieurs Embedded*. Le document client contient un document du même nom que le type d'entités ADRESSE, qui contient lui même des champs correspondants à ses attributs.

Transformation du type d'associations plusieurs-à-plusieurs

Il ne nous reste plus qu'à voir la règle de transformation du type d'associations *plusieurs-à-plusieurs*. Cette fois encore nous allons reprendre l'exemple de la figure 2.7 à laquelle nous allons ajouter ses identifiants. Il s'agit de l'exemple disant qu'une usine peut fabriquer plusieurs produits et qu'un produit peut être fabriqué par plusieurs usines. La figure 2.15 nous montre l'exemple complet ainsi que le résultat de la transformation.

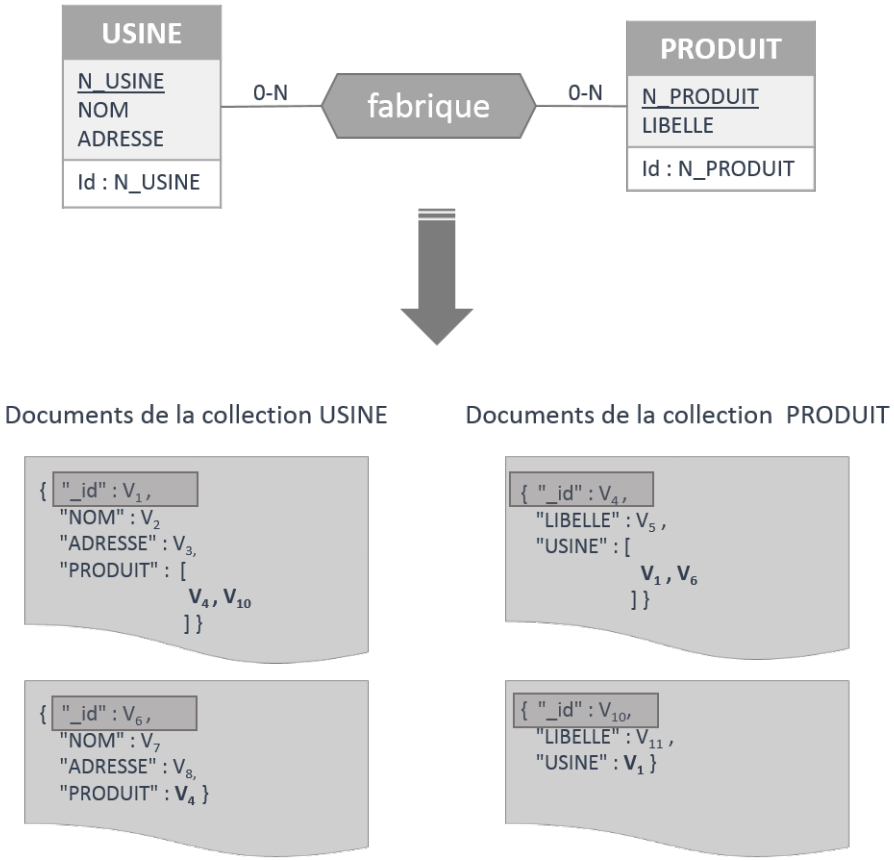


FIGURE 2.15 – Règle de transformation *plusieurs-à-plusieurs* par référencements mutuels

La règle de transformation pour ce type d'associations est assez simple. D'abord, il faut toujours appliquer la première règle sur les types d'entités. Ensuite, en plus de cela chaque document produit devra référencer les champs identifiants venant des attributs identifiants de l'autre type d'entité et ce, sous un champ qui portera le nom de l'autre type d'entité. Dans notre exemple, le document USINE comporte les références de tous les identifiants des produits qu'elle fabrique et vice-versa.

Nous venons donc de voir comment nous pouvions transformer un schéma conceptuel modélisé en Entité-Association en schéma logique correspondant à la technologie MongoDB.

Nous allons maintenant décrire l'étape de conception physique.

2.1.3 Conception physique

La conception physique a pour but de produire, sur base du schéma logique, une structure de base de données performante spécifique à la technologie choisie [Hai12]. Dans cette partie nous aborderons brièvement les moyens que l'on peut mettre en oeuvre pour avoir cette structure de base de données performante.

Il s'agit entre autre de choisir le type de cohérence, le type de réplication, les espaces de stockage, etc. Ces choix seront déterminés selon le schéma logique que nous venons de produire mais également selon les objectifs que l'on veut rencontrer, qu'il s'agisse de performance de lecture ou d'écriture, de disponibilité ou encore de fiabilité.

Performance

L'un des premiers objectifs que l'on pourrait atteindre est un gain en performance. Plusieurs facteurs peuvent contribuer au gain ou à la perte de performance. Comme nous l'avons vu dans la partie conception logique ci-dessus, nous avions par exemple le choix entre deux approches qui est d'avoir des documents référencés ou *Embedded*. Nous avons vu que selon l'approche que l'on choisit, il faudra plus ou moins de requêtes afin de retrouver les données. Généralement, qui dit plus de requêtes dit de moins bonnes performances.

Nous pouvons améliorer également la performance des requêtes en ajoutant des index où il faut. Pour reprendre les règles sur les index pour la conception physique d'une base de données relationnelle de J-L Hainaut [Hai12], il faut associer un index à chaque identifiant ainsi qu'aux clés étrangères. Pour les identifiants, c'est déjà le cas par défaut pour MongoDB par exemple. Ensuite, nous pouvons également associer un index à tout sous-document imbriqué ou référencé.

Modèle de cohérence

Comme nous l'avons déjà évoqué dans le chapitre concepts, le modèle de cohérence que l'on va choisir va avoir une incidence sur notre base de données. Par exemple, MongoDB permet de gérer son niveau de cohérence [Mon] [Sad13]. Cela se fait par l'utilisation de ces réplicats, appelés *Slaves*. Il est par exemple possible de paramétrer le système afin qu'il attende qu'une mise

à jour se soit propagée sur un nombre minimum de réplicats avant que le système réponde que la mise à jour s'est déroulée correctement. Nous pourrions aussi choisir de définir un taux de cohérence maximal qui correspondrait à ce que tous les réplicats soient cohérents à tout moment mais nous aurions dès lors des problèmes de performance.

Disponibilité

Nous pouvons faire référence ici au théorème CAP que nous avons abordé précédemment. Comme nous l'avons vu, vouloir être disponible va avoir un impact soit sur la cohérence des données et cela nous venons de le voir avec les réplicats, soit il y aura un impact sur la tolérance au partitionnement c'est-à-dire que le système ne peut pas encaisser des coupures entre noeuds en restant disponible, en même temps que d'être cohérent sur chaque noeuds.

Nous pouvons également introduire ici le principe de *load balancing* qui permet de répartir la charge des requêtes sur plusieurs réplicats.

Enfin, les technologies NoSQL permettent aussi des techniques de *Sharding* qui est de distribuer les données sur plusieurs serveurs et où chaque serveur contient un sous-ensemble unique des données.

Nous pouvons voir qu'il y a beaucoup de mécanismes que nous pouvons mettre en place afin de satisfaire nos objectifs. Le plus important est d'identifier les besoins en amont de toute conception afin de pouvoir répondre à ces objectifs.

2.2 Migration avec abandon total de SQL

Le second scénario d'adoption que nous identifions est une migration avec abandon total de SQL. Il s'agit donc ici de faire l'hypothèse que l'on souhaite migrer un système d'information existant, basé sur une base de données relationnelles, vers une plateforme NoSQL, avec l'intention de ne plus du tout utiliser la base de données relationnelle (donc de l'abandonner totalement après la migration).

Nous allons procéder par étape comme pour le scénario d'adoption précédent. La figure 2.16 montre les différentes étapes que nous allons suivre pour ce scénario. Tout d'abord, nous retrouverons le schéma de la base de données SQL. Ensuite, nous suivrons des règles de transformation similaires

à celles vues juste avant et migrerons les données transformées. Finalement, nous proposerons des exemples de transformations pour passer du langage de requête SQL à celui utilisé ici par MongoDB.

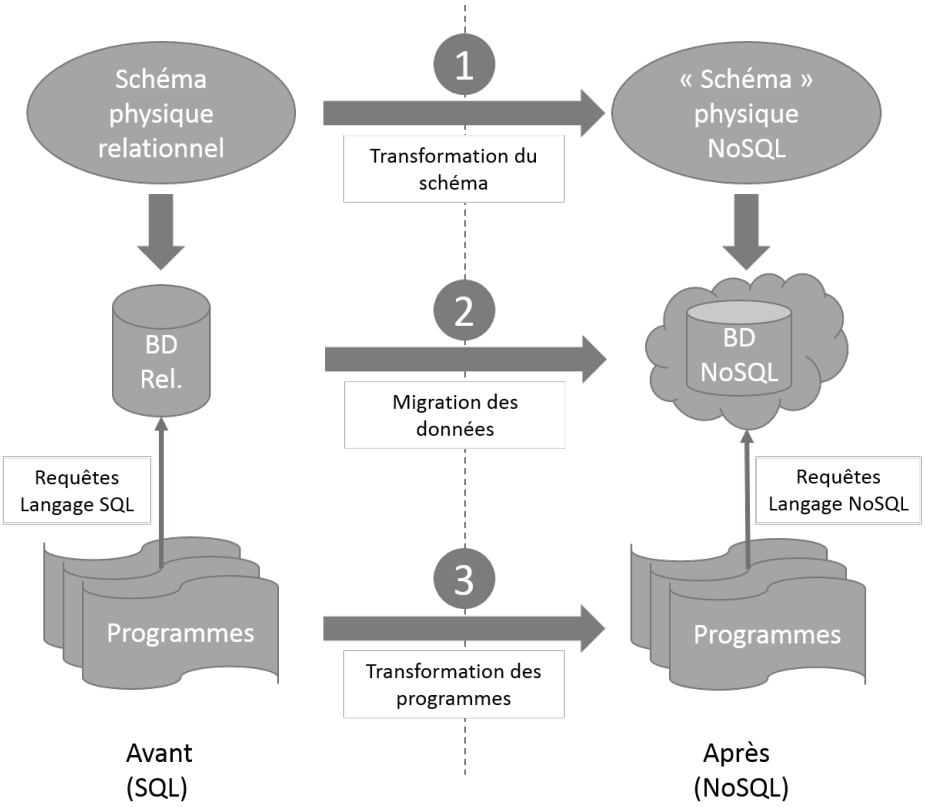


FIGURE 2.16 – Etapes du scénario de migration avec abandon total de SQL

2.2.1 Transformation du schéma

Cette première étape consiste à retrouver le schéma de la base de données SQL et de le transformer en "schéma" NoSQL. Nous devrions procéder par rétro-ingénierie afin de pouvoir retrouver le schéma conceptuel sans l'aide de l'utilisateur.

Ce mécanisme de rétro-ingénierie n'est pas nouveau [Hai12]. Il est utilisé pour produire par exemple le schéma physique, il s'agit de l'*extraction physique*. A partir de ce schéma physique, nous pouvons produire le schéma logique en faisant ce que l'on appelle la *reconstruction logique*. Et finalement, la *conceptualisation* permet de reconstituer ce qui pourrait être le schéma

conceptuel.

Dans le chapitre suivant, nous aurons l'occasion de voir plus en détail comment nous pouvons mettre en oeuvre l'extraction d'un schéma d'une base de données SQL.

Nous prendrons donc l'hypothèse ici que nous avons obtenu le schéma logique de notre base de données. Nous allons dire que notre exemple bien connu CLIENT-COMMANDE représenté à la figure 2.17 est un extrait de ce schéma que nous avons obtenu.

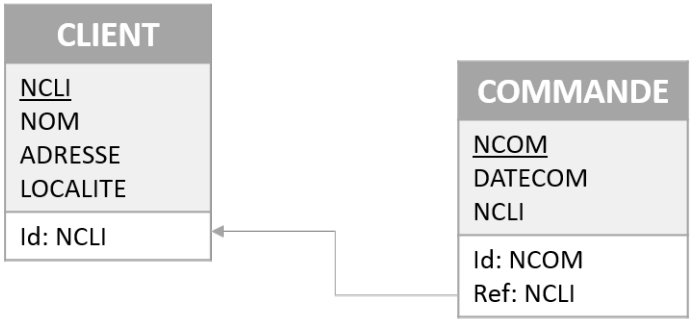


FIGURE 2.17 – Schema relationnel de la base de données CLIENT-COMMANDE

A partir de ce schéma relationnel, nous allons définir un mapping, c'est-à-dire faire correspondre les éléments du schéma de la base de données relationnelle à des éléments qui constitueront la base de données NoSQL.

Etant donné que ce mapping est directement dépendant de la technologie choisie, nous présenterons, comme convenu ci-dessus, le mapping correspondant à MongoDB.

Voici, à la figure 2.18 un rappel de mapping proposé par MongoDB [Mon].

SQL	MongoDB
database	database
table	collection
row	document
colonne	champs
clé primaire	champs _id

FIGURE 2.18 – Mapping RDBMS - Document-Oriented (MongoDB) [Mon]

Chaque table relationnelle deviendra une collection. Pour chaque ligne de ces tables correspondra un document qui possédera autant de champs que de colonnes. La clé primaire de chaque table correpondra au champ $\_id$.

Le concept de clé étrangère n'existant pas dans MongoDB, il n'y a pas d'élément exact pour faire la correspondance. Néanmoins, comme nous l'avons vu dans le scénario d'adoption précédent, nous aurons un champ qui correspondra à cette clé étrangère et qui prendra généralement comme valeur l'$\_id$ de ce qui est référencé.

La figure 2.19 montre le résultat de la transformation de schéma.

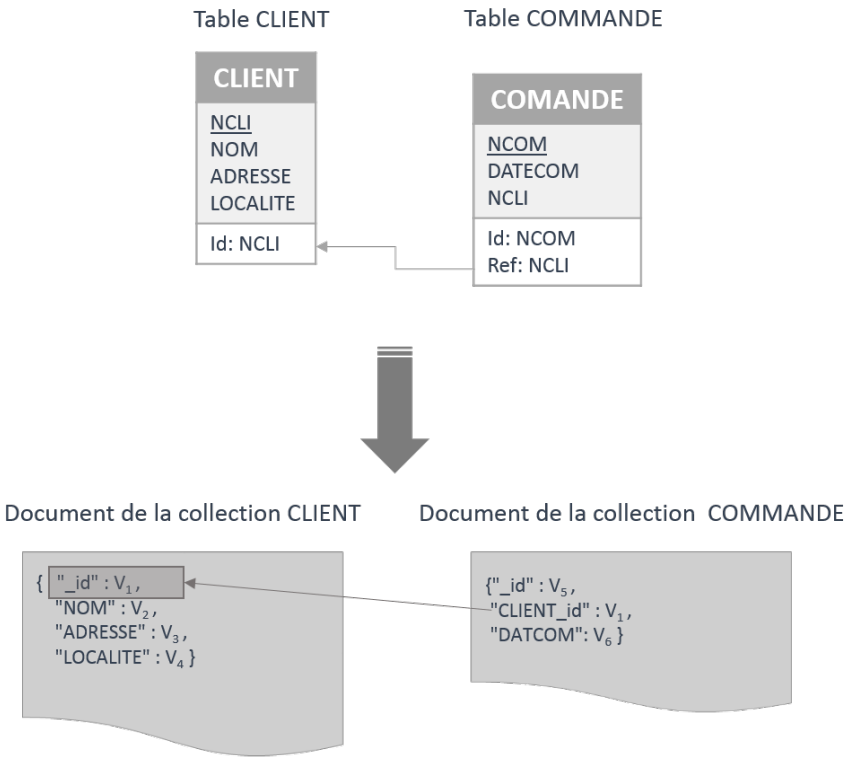


FIGURE 2.19 – Résulat du mapping SQL - NoSQL (MongoDB)

Nous avons vu précédemment qu'il y avait deux moyens de traduire le référencement pour une clé étrangère. Il s'agit des approches par référencement et *Embedded*. Dans notre exemple, nous soulignons également que ces deux approches sont possibles.

Parce que nous avons déjà vu ces approches précédemment et que les résultats dans cette méthode et dans la méthode précédente sont similaires par rapport à ces approches, nous continuerons par la suite à utiliser l'approche par référence. Notons qu'il est toujours possible de passer d'une approche à une autre via la manipulation des données.

2.2.2 Migration des données

Dans cette partie, nous allons proposer une solution pour la migration des données, de la base de données SQL à la base de données NoSQL.

Nous venons de spécifier la correspondance des éléments SQL aux éléments NoSQL. Mais il reste encore à trouver un équivalent aux instructions

qui permettent de mettre en place ces éléments. Nous allons donc voir, sur base du mapping proposé juste avant, les deux principales instructions qui seront nécessaires à la migration des données d'une base de données à l'autre. Il s'agit de la création de l'équivalent des tables, à savoir les collections. Ensuite, nous pourrons proposer une équivalence à l'instruction d'insertion des données.

Comme spécifié dans le chapitre concepts, le langage utilisé par les bases de données NoSQL orientée document et notamment MongoDB est assez riche que pour y retrouver des instructions similaires à SQL [Mon].

Pour commencer, il nous faut créer les collections. En reprenant notre exemple, il s'agira de créer une collection CLIENT et une collection COMMANDE qui correspondent aux tables. En SQL, l'instruction de création de la table CLIENT sera celle-ci :

```
create table CLIENT (   NCLI        char(10),
                        NOM         char(32),
                        ADRESSE     char(60),
                        LOCALITE    char(30),
                        primary key (NCLI));
```

En MongoDB, il n'est pas possible de créer un schéma vide. Seules les collections peuvent l'être sans contenir de documents. Nous aurons donc comme instruction MongoDB équivalente :

```
db.createCollection("CLIENT")
```

Ensuite, nous pourrons procéder à l'insertion de nouveaux clients, qui correspondront à des documents dans la collection ainsi créée. L'instruction SQL que nous devrions utiliser est :

```
insert into CLIENT (NCLI, NOM, ADRESSE, LOCALITE)
values ('B062', 'GOFFIN', '72, r. de la Gare', 'Namur');
```

L'instruction d'insertion équivalente en MongoDB sera celle-ci :

```
db.CLIENT.insert(
    {   NCLI: "B062",
        NOM: "GOFFIN",
        ADRESSE: "72, r. de la Gare",
        LOCALITE: "Namur"   }
)
```

Nous remarquons que l'instruction contient comme en orienté objet, "db." suivi du nom de la collection suivi d'un point et de l'instruction. Il en sera toujours de même pour les instructions que nous verrons par la suite.

Pour revenir à notre instruction d'insertion, notons qu'il n'est pas obligatoire de créer séparément la collection avant l'insertion des données. La création peut se faire implicitement lors de l'instruction d'insertion. Si la collection, dans ce cas-ci CLIENT, n'existe pas, elle sera créée automatiquement.

Ainsi, l'instruction d'insertion équivalente en MongoDB deviendra :

```
db.CLIENT.insert(
    {   _id: "B062",
        NOM: "GOFFIN",
        ADRESSE: "72, r. de la Gare",
        LOCALITE: "Namur"   }
)
```

Comme nous pouvons le voir, le champ NCLI est remplacé par le champ _id, afin d'imposer sa valeur comme identifiant.

Concernant la table COMMANDE dont l'instruction de création de table et d'insertion de valeur est la suivante :

```
create table COMMANDE(  NCOM        char(12),
                        DATECOM     date,
                        NCLI        char(10),
                        primary key (NCOM),
                        foreign key (NCLI) references CLIENT);

insert into COMMANDE (NCOM, DATECOM, NCLI)
values ('30186', '02/01/2008', 'B062');
```

L'équivalent en langage MongoDB se présentera comme ceci :

```
db.COMMANDE.insert(
    {   _id: "30186",
        DATECOM: "02/01/2008",
        CLIENT_id: "B062"   }
)
```

Nous pouvons voir que la clé étrangère référençant NCLI de la table CLIENT peut se faire de manière assez simple en modifiant le nom du champ

contenant la référence. Par convention, il prendra le nom de la table référencée suivi de "_id". De cette manière, nous pouvons repérer plus facilement que c'est une clé étrangère via le suffixe _id, et quelle est la collection qui est référencée via son nom.

Notons que l'instruction de création de la collection COMMANDE a volontairement été omise car celle-ci est implicite comme nous l'avons dit juste avant.

Nous venons d'aborder les deux instructions nécessaires à la création et migration des données. Afin de réellement migrer toutes les données contenues dans la base de données SQL, il est nécessaire également d'extraire toutes les données table par table avant de générer les instructions qui les insèreront dans la nouvelle base de données. Nous allons lister ces étapes.

Premièrement, sur base du schéma relationnel obtenu, nous allons générer les requêtes SQL qui vont permettre d'extraire les données de la base de données relationnelle et ceci table par table.

Deuxièmement, les données une fois extraites, vont être utilisées pour remplir les instructions NoSQL que nous allons générer également sur base du schéma relationnel.

Enfin, toutes les instructions NoSQL générées et contenant les données SQL vont être exécutées.

Il ne s'agit ici que d'une méthode que nous proposons pour la migration. Il existe bien évidemment plusieurs moyens pour y parvenir. Nous verrons dans le chapitre suivant que nous avons modestement contribué à la réalisation des étapes de migration que nous venons de citer.

2.2.3 Transformation des programmes

Afin d'être plus complet concernant notre scénario de migration, nous allons voir comment les programmes accédant à la base de données SQL devront adapter leurs requêtes pour que celle-ci soit compatible avec la base de données SQL.

Nous avons déjà vu comment nous pouvons transformer les instructions de création et d'insertion. Nous allons maintenant décrire les instructions qui concernent les modifications de données, de suppressions de données, de lectures et enfin, nous aborderons les clauses plus spécifiques comme les clauses

where, le *count*, etc.

Instruction de modification de données

Commencons par l'instruction de modification de données qui en SQL se présente sous cette forme :

```
update  CLIENT
set     ADRESSE = '29, av. de la Magne',
where   NCLI = 'B062';
```

Dans le cas de MongoDB, l'instruction *Update* contient trois paramètres dont les deux premiers sont obligatoires. Le premier est la requête qui précise sur quoi la modificaton doit porter (equivalence à la clause *where* en SQL). Le deuxième correspond à ce qu'il faut modifier (équivalence à la clause *set* en SQL). Enfin le troisième paramètre est optionnel et permet, par exemple, de définir que si aucun document ne correspond au critère de recherche, le documennt sera créé [Mon].

L'instruction de mise à jour équivalente en MongoDB se présentera comme ceci :

```
db.CLIENT.update(
   { NCLI: "B062" } , \\Critère de recherche
   { $set: { ADRESSE: "29, av. de la Magne" } } \\Modification à appliquer
)
```

Instruction de suppression de données

Voyons maintenant comment transformer les requêtes de suppression de données. En SQL, elle se présente sous cette forme :

```
delete from CLIENT
where   NCLI = 'B062';
```

Dans le cas de MongoDB, l'instruction prend deux paramètres. Le premier est le critère de recherche. Si celui-ci est vide, tout les documents de la collection spécifiée seront supprimés. Le deuxième est optionnel et permet d'arrêter la suppression au premier document rencontrant le critère de recherche. L'instruction équivalente en MongoDB se présente sous cette forme :

```
db.CLIENT.remove( { NCLI: "B062" } )
```

Instruction de lecture

Il est également possible de transformer les requêtes de lecture SQL en NoSQL. Commençons par une requête simple en SQL. Voici comment se présente une telle requête en SQL :

```
Select *
from CLIENT;
```

L'équivalence de cette instruction pour MongoDB ne contiendra aucun paramètre et dont voici le résulat :

```
db.CLIENT.find()
```

Nous allons évidemment ajouter des critères de recherche à cette instruction. Voici un exemple en SQL :

```
Select NCLI, NOM
from CLIENT
where LOCALITE = 'Namur';
```

L'instruction *find* que nous venons de voir pour MongoDB contient deux paramètres. Le premier concernera le ou les critères de recherche. Le deuxième, qui est optionnel, permet de modifier les champs qui doivent être retournés. Nous verrons que lorsque la valeur du champ est de 1, le champ est retourné (il est visible). Lorsque la valeur de champ est de 0, le champ n'est pas retourné. C'est utile lorsque par exemple, nous ne voulons pas faire apparaître le champ *_id* si elle contient un identifiant aléatoire qui pourrait polluer le résultat.

L'instruction équivalente en MongoDB se présente sous cette forme :

```
db.CLIENT.find(
    { LOCALITE: "Namur" },
    { NCLI: 1, NOM: 1, _id: 0 }
)
```

Evidemment, si nous avons choisi de faire correspondre la clé primaire SQL, qui dans notre exemple est NCLI, à l'_id MongoDB, la requête devra être adaptée. Voici le résultat :

```
db.CLIENT.find(
    { LOCALITE: "Namur" },
    { _id: 1, NOM: 1 }
)
```

Opérateurs de recherche

Nous allons maintenant aborder les équivalences que nous pouvons retrouver concernant les opérateurs de recherche. Nous nous contenterons de voir le *And*, le *OR*, le $<$, le *like* et le *count*. D'autres opérateurs étant bien sûr disponibles [Mon].

Prenons un exmple SQL un peu plus compliqué que précédemment contenant un opérateur *And* :

```
Select *
from CLIENT
where NOM = 'HANSENNE'
And   LOCALITE = 'Namur';
```

L'opérateur *And* n'existe pas en MongoDB, il suffit d'ajouter les deux critères de recherche un à coté de l'autre séparé par une virgule comme ceci :

```
db.CLIENT.find(
    { NOM: "HANSENNE",
      LOCALITE: "Namur" }
)
```

Passons maintenant aux opérateurs *OR*, $<$ et *like* dont voici un exemple qui combine le tout :

```
Select *
from CLIENT
where COMPTE < 5000
OR    NOM like '%GOF%';
```

Nous pouvons représenter l'instruction équivalente en MongoDB de cette manière :

```
db.CLIENT.find(
    { $or: [ { COMPTE: { $lt: 5000 }},
             { NOM: /GOF/ } ] }
)
```

Enfin, nous terminerons ces exemples de transformation avec l'opérateur SQL *count*.

Prenons l'exemple de requête SQL suivant :

```
Select count(*)
from CLIENT
where LOCALITE = 'Namur';
```

L'instruction SQL sera traduite en instruction MongoDB, non plus en utilisant la fonction *find* comme précédemment, mais une fonction *count* comme ceci :

```
db.CLIENT.count(
    { LOCALITE: "Namur" }
)
```

Dans cette section, nous avons proposé une méthode d'adoption en abandonnant totalement l'usage de la base de données SQL. Nous avons vu qu'il était possible moyennant quelques étapes de transformer cette base de données en base de données NoSQL en passant par la transformation du schéma, la transformation et migration des données à proprement parler et enfin par la transformation des programmes qui accèderont à cette nouvelle base de données.

Nous avons abordé à la section précédente, les notions de performance, de cohérence, de disponibilité etc. Nous ne reviendrons pas sur celles-ci dans cette partie mais elles sont toujours d'actualité pour ce type d'adoption.

Dans la section suivante, nous allons proposer une dernière méthode de migration qui est une solution de coexistence entre la base de données SQL et la base de données NoSQL.

2.3 Migration sans abandon de SQL

Les plateformes NoSQL ne doivent pas nécessairement remplacer les bases de données relationnelles classiques. Toutes deux pourraient très bien coexister dans un même environnement, chacune répondant à des problèmes différents ; ces deux types de bases de données pouvant être complémentaires.

Le terme "Co-relationnel" [MB11] peut définir ce type de coexistence. Un système co-relationnel peut être défini comme un système qui maintient et manipule des données entre des bases de données relationnelles et NoSQL [WJ14].

Un schéma du scénario que nous proposons pour cette migration est montré à la figure 2.20. Nous remarquons que les différentes étapes que nous avons vues pour la migration avec abandon de SQL, à savoir la transformation du schéma, la migration des données et la transformation des programmes restent identiques pour cette solution également. Il s'agit du même travail de transformation donc nous ne les présenterons pas une deuxième fois inutilement.

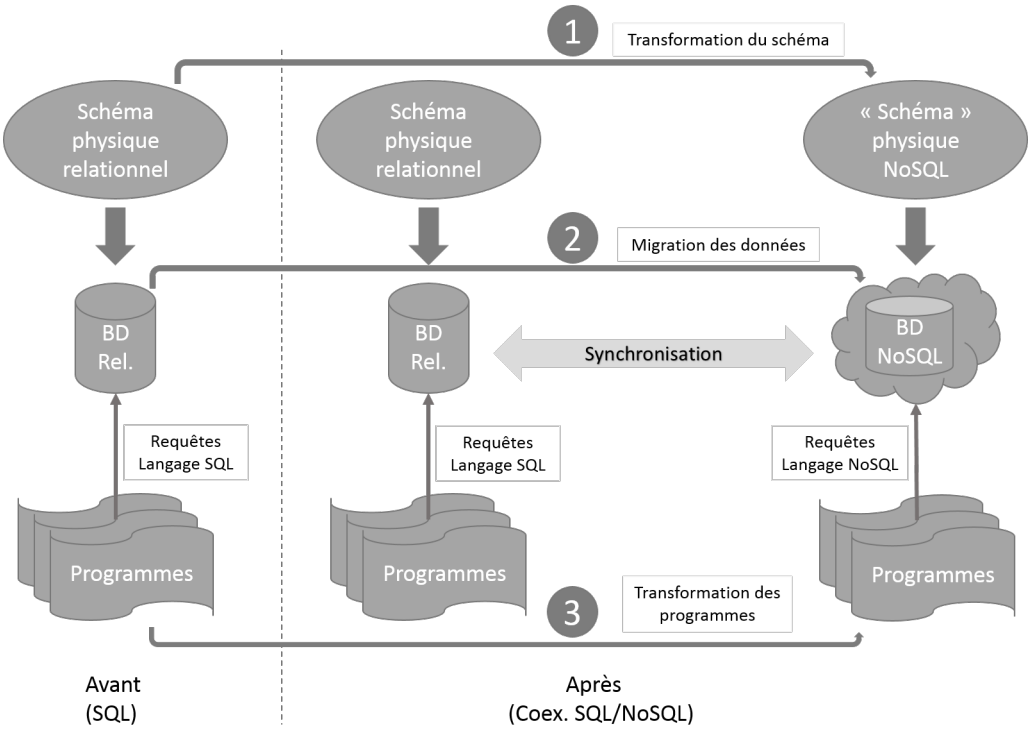


FIGURE 2.20 – Etapes de la méthodologie de migration sans abandon SQL

Comme nous pouvons le voir dans cette figure 2.20, nous proposons donc une coexistence entre SQL et NoSQL. Ce type de coexistence est intéressant à plusieurs niveaux.

Tout d'abord, nous devons différencier la coexistanece totale de la coexistence partielle. Ces deux solutions n'auront pas le même rôle à jouer.

La solution de coexistence totale est une solution où toutes les données se retrouvent dans les deux types de bases de données. Nous voyons déjà que ce n'est pas à tout moment vrai et ce quand il s'agit de gérer la cohérence

des données entre les deux plateformes. Cette solution est utile lorsque nous voulons par exemple accéder aux mêmes données mais de manières différentes. Pensons par exemple à une application qui a besoin de performance en lecture, pour afficher des articles ou des commentaires sur une page web qui est énormément visitée. Nous pourrions amener cette application à aller consulter la base de données NoSQL. Ensuite, pensons aux personnes qui laisseront des commentaires. Ces commentaires pourraient très bien être écrits directement sur la base de données SQL et une synchronisation de ceux-ci pourraient se faire sur le système NoSQL.

La solution de coexistence partielle, quant à elle, est une solution qui pourrait apporter les avantages des deux systèmes tout en faisant la distinction entre le rôle de chacun des systèmes. Les avantages que procurent les bases de données relationnelles sont généralement leur fiabilité (cfr. les transactions ACID) et la gestion de la cohérence des données. Les avantages des bases de données NoSQL ont déjà été cités auparavant, mais rappelons leur haute performance, leur tolérance vis-à-vis de l'incohérence et leur modèle de données flexible.

Nous pourrions adopter cette coexistence partielle dans un scénario où les deux bases de données manipulent un sous-ensemble de données commun, comme par exemple les données sur les clients. Mais où sont stockées également en base de données relationnelles des informations qui ont besoin de fiabilité et cohérence, pensons aux données bancaires. Avec d'un autre coté, une base de données NoSQL stockant des données moins sensibles comme par exemple des données servant à faire des statistiques globales sur des habitudes d'achats.

Maintenant que nous avons vu le contexte général où la coexistence pourrait être une solution à une problématique précise, nous pouvons pointer les problèmes liés à la synchronisation des données entre les deux technologies.

Le problème majeur pour tout système où les données sont dupliquées est la gestion de la cohérence. Comment pouvons nous nous assurer de la cohérence des données à tout moment sur ce système hybride ? Cela n'a peut-être pas d'importance car les applications qui utilisent ces données ne sont peut-être pas sensibles à cela, mais il faut à un moment ou à un autre synchroniser celle-ci.

Nous allons nous concentrer sur la synchronisation entre ces systèmes et proposer des solutions ou tout au moins donner des pistes de réflexions sur

les moyens à mettre en oeuvre pour gérer cette problématique.

Nous avons vu comment migrer les données d'une base de données à une autre dans la partie précédente consacrée à la migration avec abandon. Lorsqu'il s'agit de faire coexister l'ancien système avec le nouveau et que ceux-ci manipulent les mêmes données, il est nécessaire de garder un lien entre les mêmes données de chaque côté. Il est obligatoire de savoir quelles sont les données SQL correpondantes à ces mêmes données NoSQL. Pour cela, il va falloir mettre en place, lors de la migration, un système documentant le mapping entre les données de départ et les données migrées.

Ensuite, il faut envisager un système de journalisation des modifications qui sont effectuées au sein des deux plateformes dans le but de pouvoir rejouer ces événements. Plusieurs questions peuvent se poser au sujet de cette journalisation. Il faut définir un rythme de synchronisation. C'est-à-dire de définir le temps que nous voulons laisser les données désynchronisées, et donc le temps où l'on accepte une éventuelle incohérence. Ce temps est même appelé fenêtre d'incohérence lorsque l'on parle d'incohérence entre réplicats comme nous l'avons vu dans le chapitre sur les concepts. Le rythme que l'on va choisir va évidemment dépendre de facteurs tels que le nombre de modifications, la criticité des données, etc.

Nous pouvons soulever un deuxième problème concernant la journalisation, qui est le choix des critères de synchronisation. Celle-ci devant se faire automatiquement, il est nécessaire d'avoir des règles de synchronisation. Nous devons donc penser à mettre en place une politique de gestion de conflits. Lorsque des données se trouvent être incohérentes, il faut se baser sur cette politique. Nous pouvons envisager de définir une des deux bases de données en tant que maître, peut-être la plus fiable, afin que ce soit toujours les modifications provenant de celle-ci qui soient prioritaires. Nous pouvons également envisager de prendre la donnée la plus récente.

Enfin, cette synchronisation peut se faire dans les deux sens. C'est-à-dire SQL $\rightarrow$ NoSQL ou NoSQL $\rightarrow$ SQL. Cela nous amène à une problématique dont nous avons déjà discuté qui est la transformation de ces données en données compatibles avec l'autre système. Nous avons présenté une méthodologie pour passer des instructions SQL à à des instructions NoSQL dans la partie précédente. Il faudrait donc une méthodologie identique pour faire le chemin inverse.

Pour résumer cette partie sur la migration sans abandon de SQL, nous

avons pointé du doigt les problématiques que l'on pouvait rencontrer lors de la coexistence de deux systèmes différents. Mis à part les problèmes liés aux rôles qu'auront chacun des systèmes, la distribution des données, etc., nous avons proposé des pistes de réflexion aux problèmes de synchronisation entre ceux-ci. Rappelons, la mise en place d'un mapping liant les données des deux systèmes, la mise en place d'un mécanisme de journalisation des modifications, le paramétrage de la synchronisation en termes de rythme ou de gestion de conflits et enfin la nécessité de trouver des méthodes de transformations de données.

Chapitre 3

Contribution

Nous allons proposer une modeste contribution sous la forme d'un prototype qui a pour objectif la migration d'une base de données SQL à une base de données NoSQL.

Nous avons choisi de développer les étapes communes aux deux derniers scénarios de migration. Plus précisément, il s'agira des deux premières étapes, qui correspondent à l'étape de transformation de schéma et l'étape de migration des données.

3.1 Spécification du problème

Soit une base de données relationnelle BD_1 composées de tables $T_1, T_2, ..., T_k$. Chaque table est elle-même composée de colonnes $C_1, C_2, ..., C_m$. Pour chaque colonne d'indice i et chaque ligne d'indice j d'une table T, correspond une valeur V_{ij} , comme nous pouvons le voir à la figure 3.1.

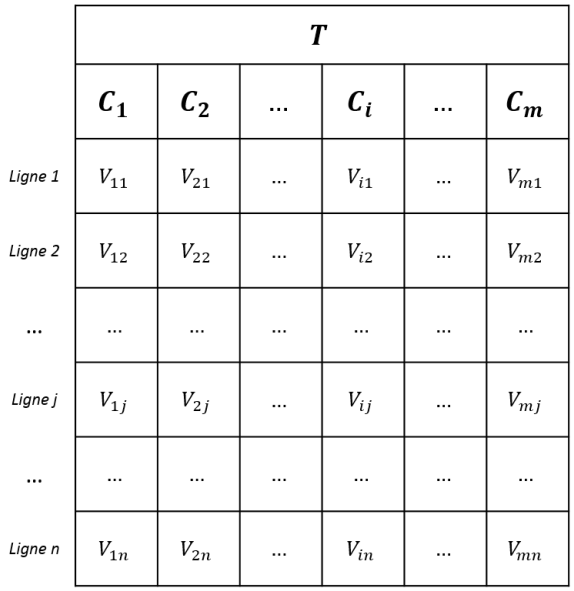


FIGURE 3.1 – Table relationnelle

Nous avons vu précédemment que selon le choix du type de base de données NoSQL que l'on fait, le modèle de données et donc le mapping cible diffèrent.

Par exemple, le schéma attendu pour une base de données NoSQL de type orienté colonne sera différent du schéma attendu pour le modèle orienté-document. Nous pouvons voir à la figure 3.2 le modèle de données selon le type orienté colonne.

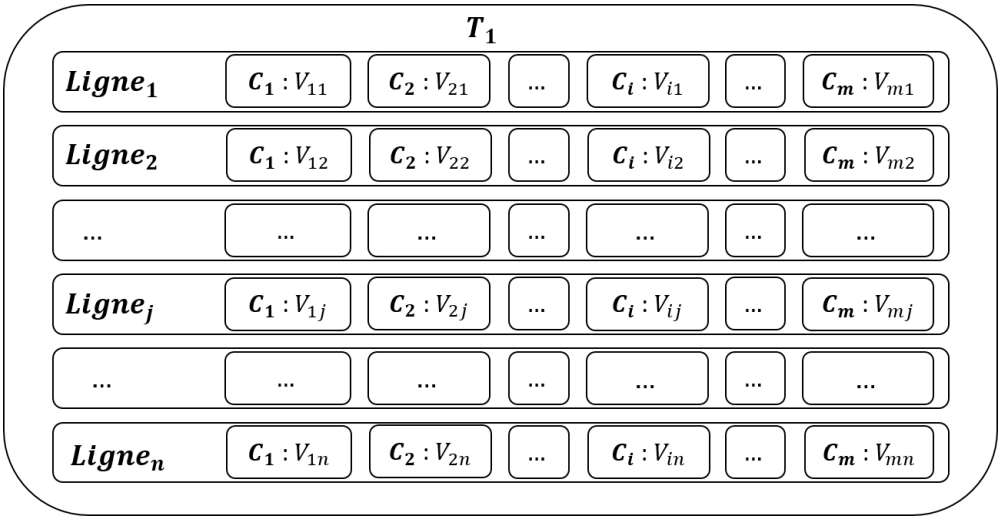


FIGURE 3.2 – Schéma orienté colonne

Selon le mapping que nous avons vu dans le chapitre des concepts, la table T_1 correspondra à une *Column Family*, les lignes de la table correspondront aux *Row* et la valeur V_{ij} contenue dans la colonne C_i correspondra aux couples *clé-valeur* de chaque ligne. Nous pouvons représenter ce modèle avec une syntaxe JSON et le résultat est donné à la figure 3.3.

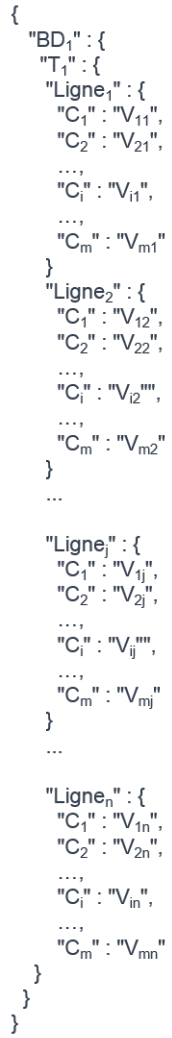


FIGURE 3.3 – Syntaxe orienté colonne

Si, en revanche, on considère le type orienté-document selon le mapping également vu au chapitre consacré aux concepts, la table T_1 correspondra à une *Collection*. La notion de ligne est quant à elle transformée en notion de document. Et la valeur V_{ij} contenue dans la colonne C_i à la ligne i correspondra aux couples *clé-valeur* de chaque document. Nous pouvons voir le résultat à la figure 3.4.

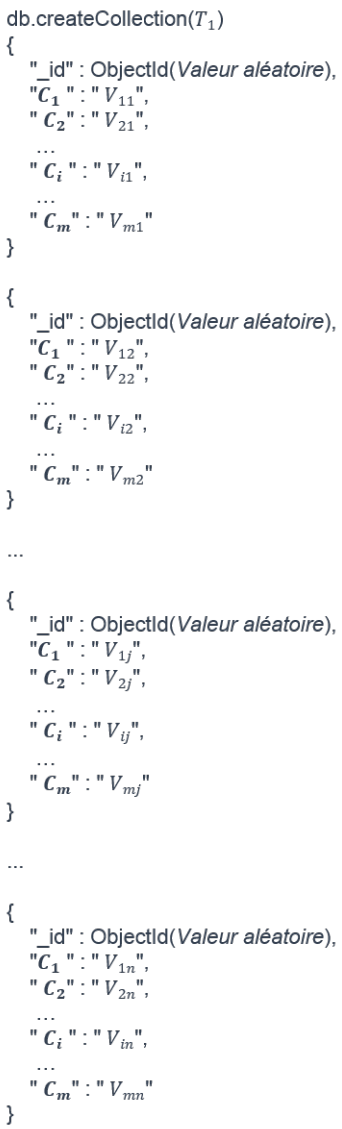


FIGURE 3.4 – Syntaxe orienté document

3.2 Approche méthodologique

Dans cette section, nous allons proposer une méthodologie pour transformer le schéma relationnel en modèle NoSQL que nous avons introduit ci-dessus. La solution proposée est une succession de cinq étapes clés. Chaque étape sera explicitée ci-dessous.

Voici les cinq étapes :

1. Extraction des métadonnées de la base de données relationnelle ;
2. Génération des requêtes SQL ;
3. Exécution des requêtes SQL ;
4. Génération des requêtes NoSQL ;
5. Exécution des requêtes NoSQL.

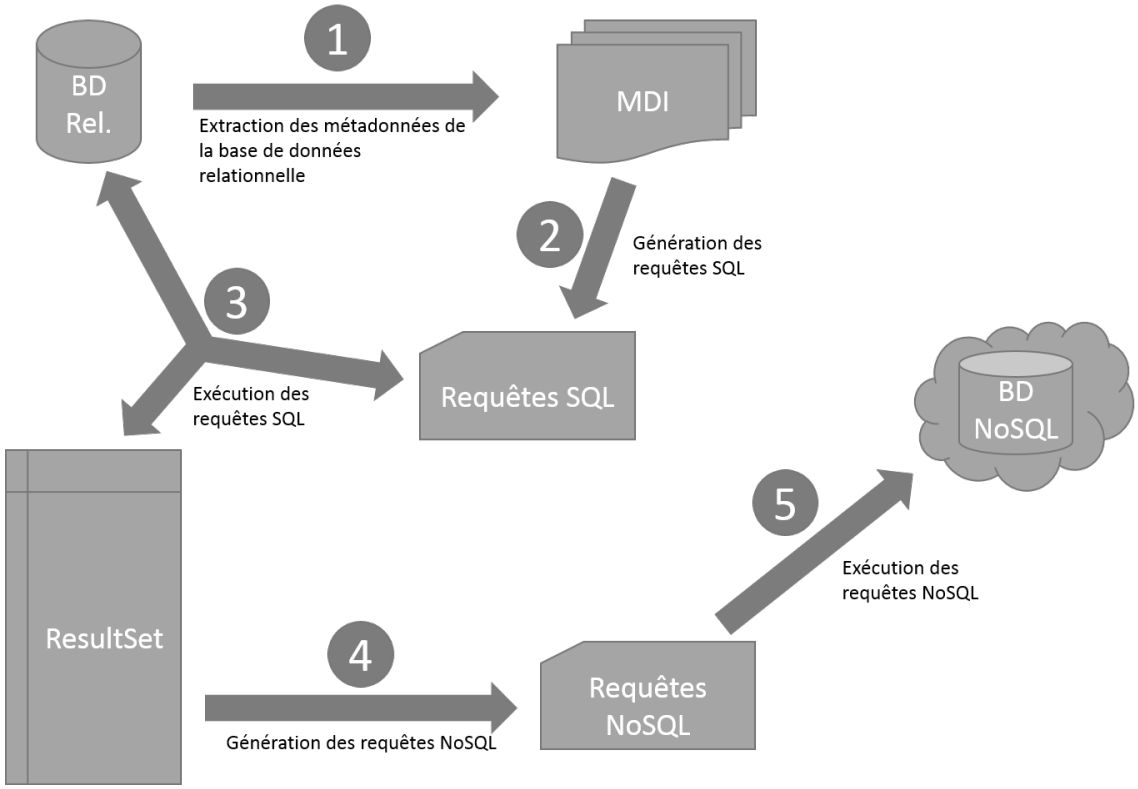


FIGURE 3.5 – Les cinq étapes clés du mapping

3.2.1 Extraction des métadonnées de la base de données relationelle

Pour pouvoir faire une transformation correcte et complète des données de la base de données relationnelle en données compatibles avec la base de données NoSQL, nous avons besoin d'un minimum d'informations concernant la base de données de départ. Nous avons notamment besoin d'extraire le schéma de la base de données SQL. Les informations ainsi collectées sont

ce que l'on appelle les métadonnées. Ces informations ne concerneront que les données relatives au schéma de la base de données.

Pour chaque table T_k de la base de données relationnelle, nous en retirerons son nom T_{Name_k} et sa/ses clé(s) primaire(s) PK_k. Pour chaque colonne C_i contenue dans une table T nous garderons son nom C_{Name_i} et son type C_{Type_i}.

Le conteneur de ces métadonnées sera appellé MDI pour MetaDataInformations. La figure 3.6 est une illustration de cette extraction de métadonnées.

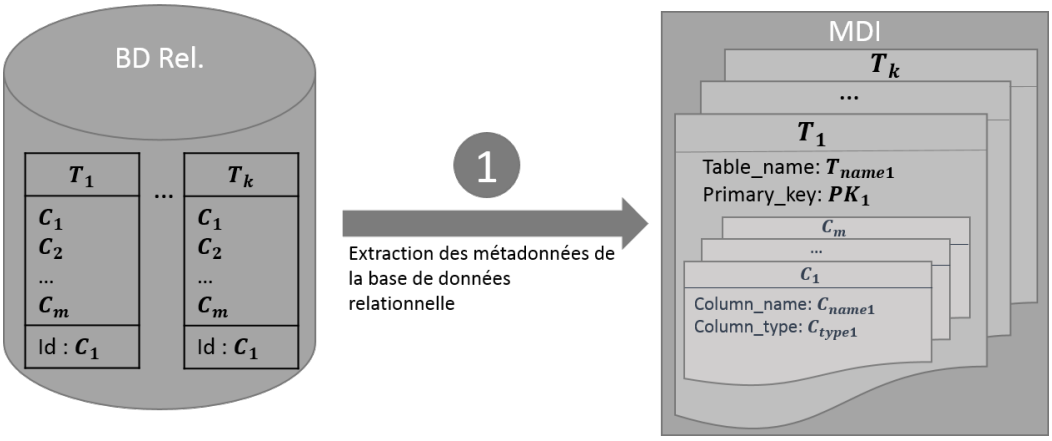


FIGURE 3.6 – Extraction des métadonnées de la base de données relationelle

Nous proposons la signature suivante pour la fonction d'extraction :

$extract\_mdi(T_n, MDI)$

Avec les notations suivantes :
– T_n : la table n ;
– MDI : le conteneur de métadonnées.

3.2.2 Génération des requêtes SQL

L'étape de génération des requêtes SQL a pour but de générer, sur base des métadonnées précédemment obtenues, les requêtes nécessaires à l'interrogation de la base de données relationnelle. Cela a pour objectif d'en extraire toutes ses valeurs.

Pour chaque table T_k contenue dans MDI, une requête SQL doit être générée.

Si l'on se base sur le schéma de la table relationnelle en figure 3.1, nous pouvons générer une requête qui aura la forme suivante :

SELECT C_1,C_2,...,C_m
FROM T

La figure 3.7 illustre l'étape de génération des requêtes SQL.

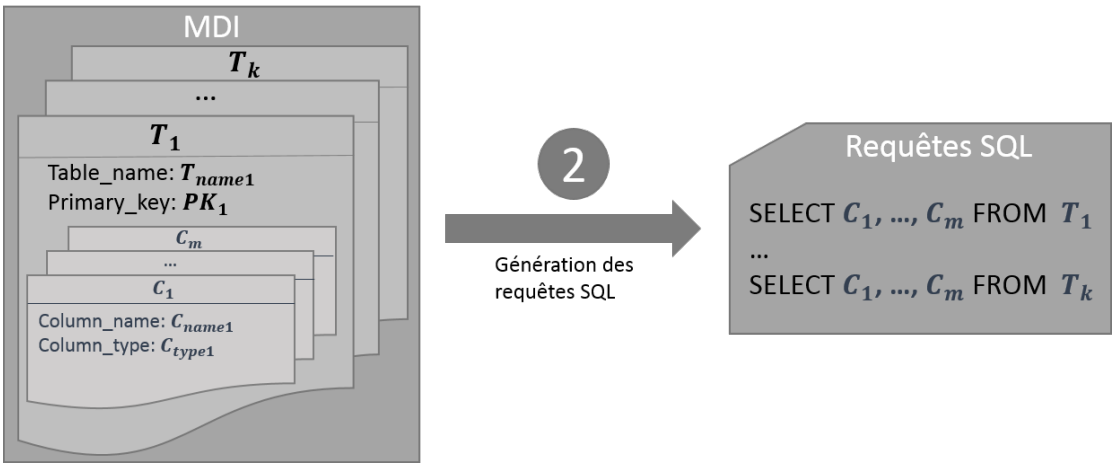


FIGURE 3.7 – Génération des requêtes SQL

Nous proposons la signature suivante pour la fonction de génération des requêtes SQL :

$$generate\_sql(MDI)$$

Avec les notations suivantes :
- MDI : le conteneur de métadonnées ;
- T_k : la table n ;
- C_i : le nom de la colonne i de la table T_k ;
- i : le nombre de colonne de T_k.

3.2.3 Exécution des requêtes SQL

Après l'étape de génération des requêtes SQL précédentes, il convient de les exécuter. Cela a pour but de pouvoir extraire toutes les données de la

base de données relationnelle.

A chaque table de la base de données relationnelle correspond donc une requête SQL qui sera exécutée. Les requêtes seront exécutées chacune à leur tour. Nous obtiendrons, pour chaque table T, un set de données set_SQL qui reprendra toutes les valeurs contenues dans T. Nous aurons donc un set de données set_SQL par table T.

La signature de la fonction d'exécution des requêtes SQL que nous proposons est la suivante :

$$execute\_sqlQuery(query) : set\_SQL$$

Avec les notations suivantes :
- $query$: la requête à exécuter ;
- set_SQL : la collection, retournée par la fonction, qui contiendra les valeurs extraites de la table T.

3.2.4 Génération des requêtes NoSQL

Jusqu'à présent, le type de base de données NoSQL n'avait aucune influence sur l'approche proposée. Pour les étapes suivantes, nous nous concentrerons, pour l'exemple, exclusivement sur la migration des données SQL vers le modèle NoSQL orienté document. Nous avons choisi pour cela la syntaxe utilisée par MongoDB [Mon].

Dans cette étape, il s'agira de transformer chaque set de données set_SQL en requêtes NoSQL qui seront par la suite exécutées. Chacune des tables deviendra comme nous l'avons vu précédemment, une collection dans ce modèle NoSQL orienté-document.

Afin d'extraire les données contenues dans un set de données set_SQL, nous aurons besoin de recourir à MDI, notamment pour connaître quel est le type de ces données à extraire. Nous aurons également besoin d'une structure de données *nosqlCollection* qui reprendra toutes les requêtes NoSQL générées ainsi que le nom de cette collection qui sera en fait le nom de la table à partir de laquelle ont été extraites ces données.

En reprenant le schéma de la table relationnelle de la Figure 3.1, la requête NoSQL générée aura, pour chaque ligne j d'une table T, cette forme :

db.T_{Name}.save({ C_1 :V_{1j} ,C_2 :V_{2j},...,C_m :V_{mj}})

La signature de la fonction de génération des requêtes NoSQL peut être celle-ci :

$nosqlQuery\_generator(set\_SQL, MDI) : nosqlCollection$

Avec les notations suivantes :
- set_SQL : le set de données SQL ;
- MDI : le conteneur de métadonnées ;
- $nosqlCollection$: la structure, retournée par la fonction, qui contiendra les requêtes NoSQL générées.

3.2.5 Exécution des requêtes NoSQL

Pour terminer, nous remarquons que nous avons toutes les informations nécessaires pour cette dernière étape. Celle-ci consiste à l'exécution des requêtes NoSQL générées à l'étape précédente.

Nous avons obtenu pour chaque table T de la base de données relationnelle, une structure de données NoSQL $nosqlCollection$ contenant après transformation les mêmes informations que ces tables T mais sous la forme de requêtes prêtes à l'emploi.

Chacune de ces structures $nosqlCollection$ sera éxécutée et donnera lieu, pour chaque, à une $Collection$ dans le modèle NoSQL orienté-document. Ces $Collections$ contiendront au final toutes les données qui était contenues au départ dans les tables de la base de données relationnelle. L'ensemble de ces $Collections$ nouvellement créées constituera la base de données NoSQL.

Enfin, nous proposons cette syntaxe pour la signature de la fonction d'exécution des requêtes NoSQL :

$execute\_nosqlQuery(nosqlCollection)$

Où $nosqlCollection$ représente la structure de données contenant les requêtes NoSQL à exécuter.

3.3 Implémentation

Ci-dessus, nous avons proposé une méthode afin de nous aider à mettre en oeuvre notre prototype. Dans cette partie, nous allons décrire les différentes structures de données que nous avons dû utiliser ainsi que décrire brièvement l'implémentation des étapes clés que nous avons identifiées.

Nous avons choisi de développer l'application de migration en Java. La base de données relationnelle est une base de données MySQL version 5.6. Elle contient notre habituelle base de données CLIENT-COMMANDE. Nous interrogerons celle-ci via l'API JDBC.

Concernant la base de données NoSQL, nous avons choisi une base de données MongoDB version 2.4.7.

Afin de visualiser les données présentes dans chacune des bases de données, nous utilisons d'une part MySQL Workbench 6.0 concernant la base de données MySQL et nous utilisons Robomongo 0.8.4 concernant la base de données MongoDB.

Nous faisons également remarquer que les classes Java, un dump de la base de données relationnelle CLIENT-COMMANDE ainsi qu'un fichier texte contenant les requêtes générées pour MongoDB sont disponibles sur le CD accompagnant ce mémoire.

3.3.1 Structures de données

Tout d'abord, nous allons présenter les structures de données dont nous avons besoin afin, notamment, d'y stocker les informations qui seront extraites de la base de données relationnelle.

– CI : La classe CI (pour *Column Informations*) permet de stocker le nom d'une colonne et son type.

– TI : La classe TI (pour *Table Informations*) permet de stocker un vecteur de CI, le nom de la ou des clés primaires.

– MDI : Nous avons utilisé une classe appelée MDI pour *MetaData Informations*. Elle permet de contenir un vecteur de TI. MDI contient donc toutes les informations nécessaires pour représenter le schéma relationnel.

– *MongoCollection* : La classe *MongoCollection* nous sert à stocker des requêtes qui ont été transformées en langage MongoDB.

– *MongoData* : Cette classe est le conteneur final de toutes les requêtes MongoDB générées. elle est composée de vecteur de *MongoCollection*.

Nous avons décrit les différentes structures de données que nous avons utilisées. La partie suivante détaillera brièvement comment nous avons développé les étapes de la partie précédente.

Extraction des métadonnées de la base de données relationnelle

Après avoir établi la connection à la base de données SQL, nous avons utilisé une classe *MDExtractor* (*MetaDataExtractor*). Cette classe contient des fonctions qui lui permettent, en recherchant dans le catalogue de la base de données SQL, notamment de découvrir les informations de chaque table de la base de données, de rechercher à l'aide du nom de ces tables, les informations sur les colonnes ainsi que les informations sur les clés primaires. Chaque information est directement placée dans la structure *MDI*.

Nous avons à ce stade toutes les informations concernant le schéma de la base de données NoSQL.

Génération des requêtes SQL

Les informations que nous venons de collecter vont pouvoir nous servir pour générer des requêtes SQL. Ces requêtes vont permettre d'extraire toutes les valeurs contenues dans les bases de données relationnelles.

Nous avons developpé une classe *MySQLQueryGenerator* contenant deux fonctions. La première va permettre, sur base d'une structure *TI* contenant les informations sur les tables, de générer une requête SQL à partir du nom de cette table et du nom de ces colonnes. La deuxième fonction, quant à elle, va simplement itérer sur un vecteur de *TI* et faire appel à la fonction de génération de requêtes. Les requêtes ainsi générées seront stockées dans un tableau de String.

Exécution des requêtes SQL et génération des requêtes NoSQL

Nous avons délibérément choisi d'assembler ces deux étapes dans une même classe pour une question de simplicité.

Nous avons trois fonctions dans cette classe appelée *MongoGenerator*. Une première fonction principale va itérer sur le tableau contenant les requêtes et fera appel à une fonction d'éxécution de requêtes afin de récupérer un *ResultSet* correspondant à la requête. Cette fonction fera aussi appel à la fonction de génération des requêtes MongoDB.

La fonction suivante a pour rôle l'exécution d'une requête sur la base de données et de renvoyer un *ResulSet*. comme nous l'avons dit plus haut, nous utiliserons l'API JDBC afin d'interroger la base de données SQL.

Enfin, la troisième est la plus intéressante car il s'agit de la fonction de génération des requêtes MongoDB. Cette fonction prend en paramètre le *ResultSet* ainsi qu'un TI qui lui correspond.

Ensuite, nous allons parcourir le *ResultSet* et à l'aide des informations contenues dans TI nous allons pouvoir connaître à tout moment le type de la valeur que nous allons recevoir via le *ResultSet*. Il est important de pouvoir connaître le type de la valeur que nous attendons car la fonction qui permet d'interroger le *ResultSet* en est dépendante. Par exemple, nous pouvons avoir *getString*(), *getDate*(), *getInt*(), etc. Nous utilisons pour cela une structure *Switch* conditionnée sur le type de la valeur afin de pouvoir utiliser la bonne fonction. Le nom de la colonne qui contenait la valeur est ensuite concaténée à la valeur elle-même. Cette concaténation est ajoutée à l'arrière de la requête.

Une fois la requête complète, nous la placerons dans la structure de données *MongoCollection*. Chaque structure *MongoCollection* sera, elle, stockée dans un vecteur contenu dans *MongoData*.
Chaque requête ainsi générée contiendra tous les noms de colonnes suivi de leur valeur pour une même ligne. Nous générerons, au final, un nombre de requêtes équivalent au nombre total de lignes de toutes les tables de la base de données SQL.

Notons que la version actuelle de notre prototype permet la génération des requêtes MongoDB, mais elle ne permet pas encore la gestion des Exécutions de celles-ci. Nous allons bien évidemment proposer des pistes d'implémentations afin que la future version puisse réaliser le processus complet de migration des données.

Nous avons donc mis en place un mécanisme d'écriture des requêtes dans

un fichier texte afin de pouvoir tout de même récuperer le résultat partiel de la migration.

Exécution des requêtes NoSQL.

Pour l'exécution des requêtes, nous allons proposer deux méthodes différentes. Mais tout d'abord, il faut savoir que l'API proposée par MongoDB *MongoJavaDriver* ne permet pas l'exécution des requêtes comme nous venons de les générer et comme nous pourrions les exécuter via le shell.

La première méthode, la plus simple à cette étape du processus, est d'utiliser une API open source appelée *Jongo*. Celle-ci permet à la manière de JDBC d'exécuter des requêtes de type String comme nous venons de générer.

La deuxième méthode est l'utilisation de l'API *MongoJavaDriver*. Celle-ci ne permet pas l'exécution de requête de type String, mais elle propose plutôt d'insérer des documents JSON.

De la même manière que nous avons généré les requêtes MongoDB ci-dessus, nous pourrions générer des documents JSON qui correpondent aux données de la base de données SQL. Il ne resterait plus qu'à exécuter les instructions d'insertions avec ces documents comme paramètres.

Améliorations

Nous terminerons la présentation de notre prototype par des pistes d'améliorations. Tout d'abord, notre prototype n'utilise pas les champs *_id* comme décrit dans le chapitre adoption. Ce manque signifie qu'il est dès lors possible d'ajouter des valeurs identifiantes en plusieurs exemplaires. Nous remarquons également que la gestion des clés étrangères n'est pas prise en charge. Néanmoins, le référencement est implicite dû à la migration des données depuis une base de données relationnelle.

Nous pouvons encore souligner que le schéma de données que nous utilisons n'est pas de type *Embedded*. Comme nous l'avons vu au chapitre précédent, il serait intéressant d'envisager ce type de schéma si nous voulons continuer sans la gestion des clés étrangères.

Enfin, bien que le prototype que nous avons développé est sans prétention, il permet de se rendre compte des mécanismes à mettre en place pour l'adop-

tion de cette technologie. Nous avons également pu remarquer qu'il n'était pas simple de trouver des méthodes structurées de développements concernant MongoDB. En revanche, nous avons pu voir une communauté plus large qu'attendue ainsi qu'une multitude de forums consacrés à ces technologies NoSQL.

Conclusion

L'arrivée des technologies NoSQL dans le domaine des bases de données est la conséquence d'une évolution concernant le traitement des données et leur stockage. Les géants du web ont permis de populariser ces technologies. Celles-ci seront amenées à gérer toujours plus d'informations à des vitesses toujours plus grandes.

L'utilisation de ces bases de données d'un nouveau genre permettent de répondre à des problématiques que les bases de données relationnelles ne pouvaient plus résoudre. Mais cette utilisation soulève des questions et des défis auxquels nous devons encore répondre. Ces problèmes, nous les avons rencontrés tout au long de la rédaction de ce mémoire. Nous pouvons citer le manque de standardisation entre plateformes, le manque d'expertise à ce sujet ou encore le manque de méthodologie au sujet de l'adoption de ces nouvelles technologies.

Résumé des contributions

Ce mémoire avait pour objectif d'aider toute personne dans sa démarche d'adoption de systèmes NoSQL. Nous avons réalisé, en plus de rappeler les concepts nécessaires à la compréhension de ce mémoire, un état de l'art des technologies NoSQL permettant d'identifier les différentes familles de bases de données NoSQL. Nous avons tenté d'identifier les cas d'utilisation les plus appropriés pour chaque famille ainsi que de donner leurs avantages et inconvénients.

Nous avons ensuite identifié et analysé trois scénarios d'adoption sur base desquels nous avons proposé des règles de transformation pour passer d'un schéma relationnel à un "schéma" NoSQL. Nous nous sommes basés pour cela, sur un exemple commun afin d'identifier et de comparer les mécanismes que l'on devrait mettre en place pour concevoir ce genre de bases de données.

Cette analyse sans prétention n'est qu'une toute petite partie d'un travail qui est encore à faire. Il est nécessaire de concevoir des méthodologies de conceptions pour ces bases de données NoSQL, comme il en existe pour les bases de données relationnelles.

Enfin, nous avons developpé un proptotype permettant d'automatiser une partie des scénarios de migration qui nous a permis d'analyser les défis soulevés par cette adoption.

Limitations

Ce mémoire est une modeste contribution à la résolution des problématiques générales que nous avons identifiées. Notre analyse des différents scénarios ne s'est focalisée que sur une mince partie des problèmes que nous pourrions rencontrer suite à l'adoption de cette technologie. En outre, le prototype est quant à lui un premier petit pas vers des systèmes de migration totalement automatisés.

Future works

Afin d'améliorer le prototype à court terme, nous devrions par exemple faire en sorte qu'il puisse transformer l'ensemble des requêtes disponibles dans MongoDB et plus seulement celles liées à l'insertion de données.

A long terme, il serait intéressant que cette application puisse produire des requêtes pour différentes technologies. Nous pourrions également allier ce genre d'applications à des outils de modélisations de bases de données comme DB-MAIN. Cela pourrait s'avérer utile lors de modélisation de base de données. Enfin, nous n'avons fait qu'évoquer les défis liés aux systèmes coexistants. Ce type de système est certainement une des pistes à explorer pour permettre d'avoir les avantages liés à ces deux technologies.

Les technologies NoSQL ne sont encore qu'à leurs balbutiements et leur utilisation ne va que s'amplifier. Il est donc nécessaire d'investir dans le développement d'outils et de méthodes pour aider à l'adoption de celles-ci.

Bibliographie

[And] Kenneth Anderson. Mysql to nosql : Data modeling challenges in supporting scalability. http ://www.infoq.com/presentations/MySQL-NoSQL-Data-Modeling/. Accessed : 2014-08-19.

[Bro] Julian Browne. Brewer's cap theorem. http ://www.julianbrowne.com/article/viewer/brewers-cap-theorem. Accessed : 2014-07-10.

[Cas] Cassandra. http ://cassandra.apache.org/.

[Cat11] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4) :12–27, 2011.

[Cod70] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6) :377–387, 1970.

[Cou] Couchdb. http ://couchdb.apache.org/.

[Dat] Oracle NoSQL Database. nosql-vs-mongodb. http ://www.oracle.com/technetwork/database/database-technologies/nosqldb/documentation/nosql-vs-mongodb-1961723.pdf. Accessed : 2014-07-17.

[Edl] Prof. Dr.Stefan Edlich. The ultimate reference for nosql databases. http ://nosql-database.org/. Accessed : 2014-07-17.

[FGC+97] Armando Fox, Steven D Gribble, Yatin Chawathe, Eric A Brewer, and Paul Gauthier. *Cluster-based scalable network services*, volume 31. ACM, 1997.

[FHM05] Michael Franklin, Alon Halevy, and David Maier. From databases to dataspaces : a new abstraction for information management. *ACM Sigmod Record*, 34(4) :27–33, 2005.

[FWC03] Joseph Fong, Hing Kwok Wong, and Zhu Cheng. Converting relational database into xml documents with dom. *Information and Software Technology*, 45(6) :335–355, 2003.

[Gaj12] Santhosh Kumar Gajendran. A survey on nosql databases. Technical report, Technical report, 2012.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2) :51–59, 2002.

[Hai12] Jean-Luc Hainaut. *Bases de données-2e éd.-Concepts, utilisation et développement : Concepts, utilisation et développement*. Dunod, 2012.

[Har] Keith W. Hare. A comparison of sqla comparison of sql and nosql databases. http ://metadata-standards.org/Document-library/Documents-by-number/WG2-N1501-N1550/WG2$_N$1537$_S$QL$_S$tandard$_a$nd$_N$oSQL$_D$atabases.Accessed : 2014 − 07 − 17.

[Heu] Nick Heudecker. Designing mongodb schemas with embedded, non-embedded and bucket structures. https ://www.openshift.com/blogs/designing-mongodb-schemas-with-embedded-non-embedded-and-bucket-structures. Accessed : 2014-08-04.

[HJ11] Robin Hecht and S Jablonski. Nosql evaluation. In *International Conference on Cloud and Service Computing*, 2011.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4) :287–317, December 1983.

[Kne12] Nataša Knez. *Primerjava relacijske in NoSQL podatkovne baze in opredelitev kriterijev za pomoč pri izbiri najprimernejše podatkovne baze*. PhD thesis, Univerza v Ljubljani, 2012.

[MB11] Erik Meijer and Gavin Bierman. A co-relational model of data for large shared data banks. *Communications of the ACM*, 54(4) :49–58, 2011.

[MCC14] KELLY MCCREARY. *Making Sense of NoSQL*. Manning Publications Co., 2014.

[Mon] Mongodb. http ://www.mongodb.org//.

[Neo] Neo4j. http ://www.neo4j.org/.

[Pok11] Jaroslav Pokorny. Nosql databases : A step to database scalability in web environment. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '11, pages 278–283, New York, NY, USA, 2011. ACM.

[Pri08] Dan Pritchett. Base : An acid alternative. *Queue*, 6(3) :48–55, May 2008.

[Raj12] Harsha Raja. Referential integrity in cloud nosql databases. 2012.

[Red12] Carter Redmond, Wilson. *Seven databases in seven weeks : a guide to modern databases and the NoSQL movement*. Dallas, 2012.

[Ria] Riak. http ://basho.com/riak/.

[Roi12] John Roijackers. *Bridging SQL and NoSQL*. PhD thesis, MSc Thesis, Eindhoven University of Technology, 2012.

[SA12] Aaron Schram and Kenneth M. Anderson. Mysql to nosql : Data modeling challenges in supporting scalability. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications : Software for Humanity*, SPLASH '12, pages 191–202, New York, NY, USA, 2012. ACM.

[Sad13] Fowler Sadalage. *NoSQL distilled : a brief guide to the emerging world of polyglot persistence*. Upper Saddle River, 2013.

[San11] Nuno Filipe Vieira dos Santos. Context storage using nosql. 2011.

[Sch] Ulrich Schiel. Integrity maintenance in advanced data bases.

[Sha] Alex Sharp. Mysql to mongo. http ://fr.slideshare.net/drumwurzel/mysql-to-mongo ? Accessed : 2014-08-25.

[Sil10] Carlos André Reis Fernandes Oliveira da Silva. *Data modeling with NoSQL : how, when and why*. PhD thesis, Tese de mestrado integrado. Engenharia Informática e Computação. Faculdade de Engenharia. Universidade do Porto., 2010.

[SM12] Oliver Schmitt and Tim A Majchrzak. Using document-based databases for medical in-formation systems in unreliable environments. 2012.

[SQL] Sqlite. http ://www.sqlite.org/.

[Sta] Mark Starkman. mongodb many-to-many relationship data modeling. http ://blog.markstarkman.com/blog/2011/09/15/mongodb-many-to-many-relationship-data-modeling/. Accessed : 2014-08-04.

[Sto10] Michael Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4) :10–11, 2010.

[Tee] Jason Tee. The three most common nosql mistakes you don't want to be making. http ://www.theserverside.com/feature/The-three-most-common-NoSQL-mistakes-you-dont-want-to-be-making. Accessed : 2014-08-19.

[Tri] Ashish Trivedi. Mapping relational databases and sql to mongodb. http ://code.tutsplus.com/articles/mapping-relational-databases-and-sql-to-mongodb–net-35650. Accessed : 2014-07-17.

[UnQ] Unql. http ://unql.sqlite.org/.

[Vog09] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1) :40–44, January 2009.

[VV01] Iraklis Varlamis and Michalis Vazirgiannis. Bridging xml-schema and relational databases : a system for generating and manipulating relational databases using valid xml documents. In *Proceedings of the 2001 ACM Symposium on Document engineering*, pages 105–114. ACM, 2001.

[WJ14] Cleve Weber-Jahnke. Corelational database maintenance – moving between the realms of sql and nosql. Technical report, 2014.

[XHZ10] Peng Xiang, Ruichun Hou, and Zhiming Zhou. Cache and consistency in nosql. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 6, pages 117–120. IEEE, 2010.

[Yon] Matt Yonkovit. How mysql and nosql coexist. http ://www.percona.com/files/presentations/percona-live/nyc-2011/PerconaLiveNYC2011-How-MySQL-and-NoSQL-Coexist.pdf. Accessed : 2014-08-19.